td1a_cenonce_session7

November 26, 2021

1 1A.algo - Programmation dynamique et plus court chemin

La programmation dynamique est une façon des calculs qui revient dans beaucoup d'algorithmes. Elle s'applique dès que ceux-ci peuvent s'écrire de façon récurrente.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

La programmation dynamique est une façon de résoudre de manière similaire une classe de problèmes d'optimisation qui vérifie la même propriété. On suppose qu'il est possible de découper le problème P en plusieurs parties P_1, P_2, … Si S est la solution optimale du problème P, alors chaque partie S_1, S_2, … de cette solution appliquée aux sous-problèmes est aussi optimale.
Par exemple, on cherche le plus court chemin $c(A, B)$ entre les villes A et B. Si celui-ci passe par la ville M alors les chemins $c(A, M) + c(M, B) = c(A, B)$ sont aussi les plus courts chemins entre les villes A, M et M, B. La démonstration se fait simplement par l'absurde : si la distance $c(A, M)$ n'est pas optimale alors il est possible de constuire un chemin plus court entre les villes A et B. Cela contredit l'hypothèse de départ.
Ces problèmes ont en règle générale une expression simple sous forme de récurrence : si on sait résoudre le problème pour un échantillon de taille n, on appelle cette solution $S(n)$ alors on peut facilement la solution $S(n + 1)$ en fonction de $S(n)$. Parfois cette récurrence va au delà : $S(n + 1) = f(S(n), S(n - 1), ..., S(0))$.

1.1 Les données

On récupère le fichier `matrix_distance_7398.txt` depuis matrix_distance_7398.zip qui contient des distances entre différentes villes (pas toutes).

[2]:
```
import pyensae.datasource
pyensae.datasource.download_data("matrix_distance_7398.zip", website  = "xd")
```

[2]:
```
['.\\matrix_distance_7398.txt']
```

On peut lire ce fichier soit avec le module pandas introduit lors de la séance 10 TD 10 : DataFrame et Matrice :

[3]:
```
import pandas
df = pandas.read_csv("matrix_distance_7398.txt", sep="\t", header=None,␣
↪names=["v1","v2","distance"])
df.head()
```

[3]:

	v1	v2	distance
0	Boulogne-Billancourt	Beauvais	85597
1	Courbevoie	Sevran	26564
2	Colombes	Alfortville	36843

```
3              Bagneux  Marcq-En-Baroeul    233455
4             Suresnes     Gennevilliers     10443
```

Le membre `values` se comporte comme une matrice, une liste de listes :

[4]:
```
matrice = df.values
matrice[:5]
```

[4]:
```
array([['Boulogne-Billancourt', 'Beauvais', 85597],
       ['Courbevoie', 'Sevran', 26564],
       ['Colombes', 'Alfortville', 36843],
       ['Bagneux', 'Marcq-En-Baroeul', 233455],
       ['Suresnes', 'Gennevilliers', 10443]], dtype=object)
```

On peut aussi utiliser le petit exemple qui a été présenté lors de la séance 4 sur les fichiers TD 4 : Modules, fichiers, expressions régulières. Les données se présente sous forme de matrice. Les deux premières colonnes sont des chaînes de caractères, la dernière est une valeur numérique qu'il faut convertir.

[5]:
```
with open ("matrix_distance_7398.txt", "r") as f :
    matrice = [ row.strip(' \n').split('\t') for row in f.readlines() ]
for row in matrice:
    row[2] = float(row[2])
print(matrice[:5])
```

```
[['Boulogne-Billancourt', 'Beauvais', 85597.0], ['Courbevoie', 'Sevran',
26564.0], ['Colombes', 'Alfortville', 36843.0], ['Bagneux', 'Marcq-En-Baroeul',
233455.0], ['Suresnes', 'Gennevilliers', 10443.0]]
```

Chaque ligne définit un voyage entre deux villes effectué d'une traite, sans étape. Les accents ont été supprimés du fichier.

1.2 Exercice 1

Construire la liste des villes sans doublons.

[6]:
```

```

1.3 Exercice 2

Constuire un dictionnaire `{ (a,b) : d, (b,a) : d }` où `a,b` sont des villes et `d` la distance qui les sépare ?
On veut calculer la distance entre la ville de `Charleville-Mezieres` et `Bordeaux` ? Est-ce que cette distance existe dans la liste des distances dont on dispose ?

[7]:
```

```

1.4 Algorithme du plus court chemin

On créé un tableau `d[v]` qui contient ou contiendra la distance optimale entre les villes `v` et `Charleville-Mezieres`. La valeur qu'on cherche est `d['Bordeaux']`. On initialise le tableau comme suit :

- `d['Charleville-Mezieres'] = 0`
- `d[v] = infini` pour tout $v \neq$ `'Charleville-Mezieres'`.

1.5 Exercice 3

Quelles sont les premières cases qu'on peut remplir facilement ?

[8]:

1.6 Exercice 4

Soit une ville v et une autre w, on s'aperçoit que $d[w] > d[v] + dist[w, v]$. Que proposez-vous de faire ? En déduire un algorithme qui permet de déterminer la distance la plus courte entre Charleville-Mezieres et Bordeaux.
Si la solution vous échappe encore, vous pouvez vous inspirer de l'Algorithme de Djikstra.

[9]:

1.7 La répartition des skis

Ce problème est un exemple pour lequel il faut d'abord prouver que la solution vérifie une certaine propriété avant de pouvoir lui appliquer une solution issue de la programmation dynamique.
$N = 10$ skieurs rentrent dans un magasins pour louer 10 paires de skis (parmi $M > N$). On souhaite leur donner à tous une paire qui leur convient (on suppose que la taille de la paire de skis doit être au plus proche de la taille du skieurs. On cherche donc à minimiser :
$\arg\min_\sigma \sum_{i=1}^{N} \left|t_i - s_{\sigma(i)}\right|$
Où σ est un ensemble de N paires de skis parmi M (un arrangement pour être plus précis).
A première vue, il faut chercher la solution du problème dans l'ensemble des arrangements de N paires parmi M. Mais si on ordonne les paires et les skieurs par taille croissantes : $t_1 \leqslant t_2 \leqslant ... \leqslant t_N$ (tailles de skieurs) et $s_1 \leqslant s_2 \leqslant ... \leqslant s_M$ (tailles de skis), résoudre le problème revient à prendre les skieurs dans l'ordre croissant et à les placer en face d'une paire dans l'ordre où elles viennent. C'est comme si on insérait des espaces dans la séquence des skieurs sans changer l'ordre :

t_1		t_2	t_3			t_4	...	t_{N-1}		t_N	
s_1	s_2	s_3	s_4	s_5	s_6	s_7	...	s_{M-3}	s_{M-2}	s_{M-1}	s_M

[10]:

1.8 Exercice facultatif

Il faut d'abord prouver que l'algorithme suggéré ci-dessus permet bien d'obtenir la solution optimale.

1.9 Exercice 5

Après avoir avoir trié les skieurs et les paires par tailles croissantes. On définit :
$p(n, m) = \sum_{i=1}^{n} \left|t_i - s_{\sigma^*_m(i)}\right|$
Où σ^*_m est le meilleur choix possible de n paires de skis parmi les m premières. Exprimer $p(n, m)$ par récurrence (en fonction de $p(n, m-1)$ et $p(n-1, m-1)$. On suppose qu'un skieur sans paire de ski correspond au cas où la paire est de taille nulle.

[11]:

1.10 Exercice 6

Ecrire une fonction qui calcule l'erreur pour la distribution optimale ? On pourra choisir des skieurs et des paires de tailles aléatoires par exemple.

[12]:
```
import random
skieurs = [ random.gauss(1.75, 0.1) for i in range(0,10) ]
paires  = [ random.gauss(1.75, 0.1) for i in range(0,15) ]
skieurs.sort()
paires.sort()
```

```
print(skieurs)
print(paires)
```

1.11 Exercice 7

Quelle est la meilleure distribution des skis aux skieurs ?

[13]:

1.12 Exercice 8

Quels sont les coûts des deux algorithmes (plus court chemin et ski) ?

[14]:

1.13 Prolongements : degré de séparation sur Facebook

Le plus court chemin dans un graphe est un des algorithmes les plus connus en programmation. Il permet de déterminer la solution en un coût **polynômial** - chaque itération est en $O(n^2)$. La programmation dynamique caractèrise le passage d'une vision combinatoire à une compréhension récursif du même problème. Dans le cas du plus court chemin, l'approche combinatoire consiste à énumérer tous les chemins du graphe. L'approche dynamique consiste à démontrer que la première approche combinatoire aboutit à un calcul très redondant. On note $e(v,w)$ la matrice des longueurs des routes, $e(v,w) = \infty$ s'il n'existe aucune route entre les villes v et w. On suppose que $e(v,w) = e(w,v)$. La construction du tableau `d` se définit de manière itérative et récursive comme suit :

Etape 0

$d(v) = \infty, \forall v \in V$

Etape n

$$d(v) = \left\{ \begin{array}{ll} 0 & si\ v = v_0 \\ \min\{d(w) + e(v,w) \,|\, w \in V\} & sinon \end{array} \right. \quad \text{où } v_0 = \texttt{'Charleville-Mezieres'}$$

Tant que l'étape n continue à faire des mises à jour ($\sum_v d(v)$ diminue), on répète l'étape n. Ce même algorithme peut être appliqué pour déterminer le degré de séparation dans un réseau social. L'agorithme s'applique presque tel quel à condition de définir ce que sont une ville et une distance entre villes dans ce nouveau graphe. Vous pouvez tester vos idées sur cet exemple de graphe Social circles: Facebook. L'algorithme de Dikjstra calcule le plus court chemin entre deux noeuds d'un graphe, l'algorithme de Bellman-Ford est une variante qui calcule toutes les distances des plus courts chemin entre deux noeuds d'un graphe.

[15]:
```
import pyensae.datasource
files = pyensae.datasource.download_data("facebook.tar.gz",website="http://snap.
 ↪stanford.edu/data/")
fe = [ f for f in files if "edge" in f ]
fe
```

Il faut décompresser ce fichier avec 7zip si vous utilisez `pysense < 0.8`. Sous Linux (et Mac), il faudra utiliser une commande décrite ici tar.

[16]:
```
import pandas
df = pandas.read_csv("facebook/1912.edges", sep=" ", names=["v1","v2"])
print(df.shape)
df.head()
```

[17]: