CoursMagistral_7

November 5, 2019

1 Table of Contents

2.8 Conclusion

2 ALGO1 : Introduction à l'algorithmique

- Page du cours : https://perso.crans.org/besson/teach/info1_algo1_2019/
- Magistère d'Informatique de Rennes - ENS Rennes - Année 2019/2020
- Intervenants :
  - Cours : Lilian Besson
  - Travaux dirigés : Raphaël Truffet
- Références :
  - Open Data Structures

3 Cours Magistral 7

- Ce cours traite des algorithmes par programmation dynamique.

3.1 Optimisation de chaînes de multiplications de matrices

[1]:
```python
# https://github.com/jilljenn/tryalgo/blob/master/tryalgo/matrix_chain_mult.py
def matrix_mult_opt_order(M):
    """Matrix chain multiplication optimal order

    :param M: list of matrices
    :returns: matrices opt, arg, such that opt[i][j] is the optimal number of
              operations to compute M[i] * ... * M[j] when done in the order
              (M[i] * ... * M[k]) * (M[k + 1] * ... * M[j]) for k = arg[i][j]
    :complexity: :math:`O(n^2)`
    """
    n = len(M)
    r = [len(Mi) for Mi in M]
    c = [len(Mi[0]) for Mi in M]
    opt = [[0 for j in range(n)] for i in range(n)]
    arg = [[None for j in range(n)] for i in range(n)]
    for j_i in range(1, n):   # loop on i, j of increasing j - i = j_i
        for i in range(n - j_i):
            j = i + j_i
            opt[i][j] = float('inf')
            for k in range(i, j):
                alt = opt[i][k] + opt[k + 1][j] + r[i] * c[k] * c[j]
                if opt[i][j] > alt:
                    opt[i][j] = alt
                    arg[i][j] = k
    return opt, arg
```

[4]:
```python
def matrix_chain_mult(M):
    """Matrix chain multiplication
```

```
    :param M: list of matrices
    :returns: M[0] * ... * M[-1], computed in time optimal order
    :complexity: whatever is needed by the multiplications
    """
    opt, arg = matrix_mult_opt_order(M)
    return apply_order(M, arg, 0, len(M)-1)
```

[5]:
```
def apply_order(M, arg, i, j):
    # --- multiply matrices from M[i] to M[j] included
    if i == j:
        return M[i]
    else:
        k = arg[i][j]        # --- follow placement of parentheses
        A = apply_order(M, arg, i, k)
        B = apply_order(M, arg, k + 1, j)
        row_A = range(len(A))
        row_B = range(len(B))
        col_B = range(len(B[0]))
        return [[sum(A[a][b] * B[b][c] for b in row_B)
                 for c in col_B] for a in row_A]
```

Et juste pour avoir une comparaison honnête avec les multiplications de Numpy, on écrit les mêmes fonctions mais utilisant `A @ B` (`numpy.dot`) pour multiplier les matrices :

[40]:
```
def apply_order_numpy(M, arg, i, j):
    # --- multiply matrices from M[i] to M[j] included
    if i == j:
        return M[i]
    else:
        k = arg[i][j]        # --- follow placement of parentheses
        A = apply_order_numpy(M, arg, i, k)
        B = apply_order_numpy(M, arg, k + 1, j)
        return A @ B
```

[41]:
```
def matrix_chain_mult_numpy(M):
    """Matrix chain multiplication

    :param M: list of matrices
    :returns: M[0] * ... * M[-1], computed in time optimal order
    :complexity: whatever is needed by the multiplications
    """
    opt, arg = matrix_mult_opt_order(M)
    return apply_order_numpy(M, arg, 0, len(M)-1)
```

3.1.1 Par exemple : avec A_1 de taille $(10, 100)$, A_2 de taille $(100, 5)$ et A_3 de taille $(5, 50)$.

[42]:
```
import numpy as np
```

[43]:
```
A1 = np.random.randint(-1000, 1000, size=(10, 100))
A2 = np.random.randint(-1000, 1000, size=(100, 5))
A3 = np.random.randint(-1000, 1000, size=(5, 50))
```

[44]:
```
np.shape(A1)
np.shape(A2)
np.shape(A3)
```

[44]: (10, 100)

[44]: (100, 5)

[44]: (5, 50)

[45]:
```
prod0 = A1 @ A2 @ A3  # let Python decide the order !

prod1 = (A1 @ A2) @ A3
assert np.alltrue(prod0 == prod1)

prod2 = A1 @ (A2 @ A3)
assert np.alltrue(prod0 == prod2)

assert np.alltrue(prod1 == prod2)
```

[27]:
```
np.shape(prod1)
np.shape(prod2)
```

[27]: (10, 50)

[27]: (10, 50)

[28]:
```
%timeit A1 @ A2 @ A3
%timeit (A1 @ A2) @ A3
%timeit A1 @ (A2 @ A3)
```

```
15.5 µs ± 841 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
15.5 µs ± 1.93 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
118 µs ± 4.73 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Python utilise l'associativité à gauche, par défaut, comme le montre les calculs ci dessus : `A1 @ A2 @ A3` est calculé comme `(A1 @ A2) @ A3`. On peut le vérifier avec un autre exemple :

[32]:
```
10 - 3 - 2
(10 - 3) - 2  # x - y - z = (x - y) - z oui
```

```
10 - (3 - 2)  # x - y - z != x - (y - z) non
```

[32]: 5

[32]: 5

[32]: 9

[46]:
```
M = [A1, A2, A3]
%timeit matrix_chain_mult(M)
%timeit matrix_chain_mult_numpy(M)
```

```
12.1 ms ± 371 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
38.3 µs ± 1.16 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

[29]:
```
print("Finding the optimal order to multiply chain of matrices of dimensions")
print([np.shape(Mi) for Mi in M])
opt, arg = matrix_mult_opt_order(M)
print("opt =", opt)
print("arg =", arg)
```

```
Finding the optimal order to multiply chain of matrices of dimensions
[(10, 100), (100, 5), (5, 50)]
opt = [[0, 5000, 7500], [0, 0, 25000], [0, 0, 0]]
arg = [[None, 0, 1], [None, None, 1], [None, None, None]]
```

Ce vecteur `arg` nous dit qu'il faut d'abord multiplier `M[0]` avec `M[1]` (soit `A1 @ A2`), puis multiplier le résultat avec `M[2]` (soit `(A1 @ A2) @ A3` comme trouvé ci dessus).

3.1.2 Par exemple : avec A_1 de taille $(10, 100)$, A_2 de taille $(100, 5)$ et A_3 de taille $(5, 50)$ et A_4 de taille $(50, 200)$.

[47]:
```
A1 = np.random.randint(-1000, 1000, size=(10, 100))
A2 = np.random.randint(-1000, 1000, size=(100, 5))
A3 = np.random.randint(-1000, 1000, size=(5, 50))
A4 = np.random.randint(-1000, 1000, size=(50, 200))
```

[48]:
```
prod0 = A1 @ A2 @ A3 @ A4  # let Python decide the order !

prod1 = ((A1 @ A2) @ A3) @ A4
assert np.alltrue(prod0 == prod1)

prod2 = (A1 @ (A2 @ A3)) @ A4
assert np.alltrue(prod0 == prod2)

prod3 = (A1 @ A2) @ (A3 @ A4)
assert np.alltrue(prod0 == prod3)
```

```
prod4 = A1 @ ((A2 @ A3) @ A4)
assert np.alltrue(prod0 == prod4)

prod5 = A1 @ (A2 @ (A3 @ A4))
assert np.alltrue(prod0 == prod5)
```

[49]:
```
np.shape(prod1)
np.shape(prod2)
np.shape(prod3)
np.shape(prod4)
np.shape(prod5)
```

[49]: (10, 200)

[49]: (10, 200)

[49]: (10, 200)

[49]: (10, 200)

[49]: (10, 200)

[50]:
```
%timeit A1 @ A2 @ A3 @ A4

%timeit ((A1 @ A2) @ A3) @ A4
%timeit (A1 @ (A2 @ A3)) @ A4
%timeit (A1 @ A2) @ (A3 @ A4)
%timeit A1 @ ((A2 @ A3) @ A4)
%timeit A1 @ (A2 @ (A3 @ A4))
```

```
201 µs ± 8.84 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
196 µs ± 2.94 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
319 µs ± 5.78 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
139 µs ± 5.27 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
2.2 ms ± 241 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
673 µs ± 22.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[54]:
```
M = [A1, A2, A3, A4]
%timeit matrix_chain_mult(M)
%timeit matrix_chain_mult_numpy(M)
```

```
80.4 ms ± 9.9 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
141 µs ± 12.1 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

[55]:
```
print("Finding the optimal order to multiply chain of matrices of dimensions")
print([np.shape(Mi) for Mi in M])
opt, arg = matrix_mult_opt_order(M)
```

```
print("opt =", opt)
print("arg =", arg)
```

```
Finding the optimal order to multiply chain of matrices of dimensions
[(10, 100), (100, 5), (5, 50), (50, 200)]
opt = [[0, 5000, 7500, 65000], [0, 0, 25000, 150000], [0, 0, 0, 50000], [0, 0,
0, 0]]
arg = [[None, 0, 1, 1], [None, None, 1, 1], [None, None, None, 2], [None, None,
None, None]]
```

3.2 Plus longue sous séquence commune

Je ne rentre pas dans les détails, allez voir sur Internet, ou notamment des développements préparés par des élèves de l'ENS de Rennes candidat-e-s à l'agrégation de mathématiques option informatique : ici ou ici.

[13]:
```
def longest_common_subsequence(x, y):
    """longest common subsequence

    Dynamic programming

    :param x:
    :param y: x, y are lists or strings
    :returns: longest common subsequence in form of a string
    :complexity: `O(|x|*|y|)`
    """
    n = len(x)
    m = len(y)
    #                       -- compute optimal length
    A = [[0 for j in range(m + 1)] for i in range(n + 1)]
    for i in range(n):
        for j in range(m):
            if x[i] == y[j]:
                A[i + 1][j + 1] = A[i][j] + 1
            else:
                A[i + 1][j + 1] = max(A[i][j + 1],  A[i + 1][j])
    #                       -- extract solution
    sol = []
    i, j = n, m
    while A[i][j] > 0:
        if A[i][j] == A[i - 1][j]:
            i -= 1
        elif A[i][j] == A[i][j - 1]:
            j -= 1
        else:
            i -= 1
```

```
            j -= 1
            sol.append(x[i])
    return ''.join(sol[::-1])    # inverse solution
```

Exemples :

[14]:
```
longest_common_subsequence("BABAR", "ACAB")
```

[14]:
```
'AB'
```

[15]:
```
longest_common_subsequence("J'aime bien les oiseaux !", "Les oiseaux sont des␣
↪animaux à protéger.")
```

[15]:
```
'e ien es iaux '
```

[16]:
```
longest_common_subsequence("Je suis amoureux de la vie", "Tu n'es pas amoureux␣
↪d'elle ?")
```

[16]:
```
'e s amoureux del '
```

3.3 Plus longue sous séquence croissante

On traite le sous-problème suivant : $L(k)$ est la longueur de la plus longue sous séquence croissante de $[a_1, \ldots, a_k]$.

$$L(1) = 1$$
$$\forall k > 0, L(k+1) = \max(L(k), \max(L(i) + 1 : a[sol[i][-1]] <= a[k+1], i <= k))$$

C'est donc très facile d'écrire la solution :

[62]:
```
def longest_increasing_sequence(a):
    n = len(a)
    L = [0] * n
    sol = [[i] for i in range(n)]
    # L[i] = length of longer increasing sequence of a[0]..a[i]
    L[0] = 1
    sol[0] = [0]  # just <a[0]> is increasing for a[0]..a[0]
    for j in range(1, n):
        # L[j] = max(L[j-1], max(L[i]+1 for i < j such that a[sol[i][-1]] <=␣
↪a[j]))
        ell = 1
        for i in range(j):
            if a[sol[i][-1]] <= a[j] and L[i] + 1 > ell:
                ell = L[i] + 1
                sol[j] = sol[i] + [j]  # sol of a[0]..a[i] + a[j]
        if ell < L[j-1]:
```

```
            ell = L[j-1]
            sol[j] = sol[i][::]
        L[j] = ell
    return [a[i] for i in sol[n-1]], sol[n-1], max(L)
```

Avec l'exemple utilisé en cours :

[59]:
```
sequence = [19, 1, 9, 27, 26]
```

[61]:
```
for i in range(1, len(sequence) + 1):
    print("Pour la séquence", sequence[:i], "la plus longue sous séquence␣
↪croissante est", longest_increasing_sequence(sequence[:i]))
```

```
Pour la séquence [19] la plus longue sous séquence croissante est ([19], [0], 1)
Pour la séquence [19, 1] la plus longue sous séquence croissante est ([1], [1],
1)
Pour la séquence [19, 1, 9] la plus longue sous séquence croissante est ([1, 9],
[1, 2], 2)
Pour la séquence [19, 1, 9, 27] la plus longue sous séquence croissante est ([1,
9, 27], [1, 2, 3], 3)
Pour la séquence [19, 1, 9, 27, 26] la plus longue sous séquence croissante est
([1, 9, 26], [1, 2, 4], 3)
```

3.4 Algorithme de Bellman-Ford

Chemin de k sommets : on considère le sous-problème $P(k)$ suivant : chercher la longueur des plus courts chemins comportant au plus k sommets ($k \in \{1, \dots, n\}$).

On fixe l'origine des chemins, s ; on note $\delta(k, t)$ la longueur minimale d'un chemin d'au plus k arcs de s à t. On a la relation de récurrence suivante :

$$\delta(0, t) = \begin{cases} 0 & \text{si } s = t \\ \infty & \text{sinon.} \end{cases}$$

$$\delta(k+1, t) = \min(\delta(k, t), \min\{\delta(k, u) + w(u, t), u \in S\})$$

Cela donne l'algorithme suivant, qui fixe $s \in S$ le sommet de départ. L'algorithme de Bellman-Ford s'exécute en mémoire $\mathcal{O}(|S|)$ et en temps $\mathcal{O}(|S| * |A|)$.

Donc avec tous les sommets $s \in S$ différents, cela donne $\mathcal{O}(|S|^2|A|) \leq \mathcal{O}(|S|^4)$.

[63]:
```
def bellman_ford(graph, weight, source=0):
    """ Single source shortest paths by Bellman-Ford

    :param graph: directed graph in listlist or listdict format
    :param weight: can be negative.
                   in matrix format or same listdict graph
    :returns: distance table, precedence table, bool
```

```
    :explanation: bool is True if a negative circuit is
                  reachable from the source, circuits
                  can have length 2.
    :complexity: `O(|S|*|A|)`
    """
    n = len(graph)
    dist = [float('inf')] * n
    prec = [None] * n
    dist[source] = 0
    # this loop is in O(|S|) nb of vertex
    for nb_iterations in range(n):
        changed = False
        # these two loops are in O(\sum_u degree(u)) = O(|A|) nb of edges
        for node in range(n):
            for neighbor in graph[node]:
                alt = dist[node] + weight[node][neighbor]
                if alt < dist[neighbor]:
                    dist[neighbor] = alt
                    prec[neighbor] = node
                    changed = True
        if not changed:                    # fixed point
            return dist, prec, False
    return dist, prec, True
```

Un exemple de graphe, comme celui utilisé dans la page Wikipédia :

[26]:
```
oo = float('+inf')

weight = [
    [0, oo, -2, oo],
    [4, 0, 3, oo],
    [oo, oo, 0, 2],
    [oo, -1, oo, 0],
]
```

[27]:
```
n = len(weight)
graph = [
    [ v for v in range(n) if weight[u][v] < oo ]
    for u in range(n)
]
```

[28]:
```
bellman_ford(graph, weight)
```

[28]:
```
([0, -1, -2, 0], [None, 3, 0, 2], False)
```

3.5 Algorithme de Floyd-Warshall

Chemin empruntant les sommets $\{1, \dots, k\}$ On suppose que $S = \{1, \dots, n\}$ (quitte à ré-étiqueter les sommets), on s'intéresse au sous-problème $P(k)$ suivant~: chercher la longueur des plus courts chemins dont les sommets intermédiaires sont dans $\{1, \dots, k\}$.

On note maintenant $\delta(k, s, t)$ la longueur minimale d'un chemin de s à t dont les sommets intermédiaires sont dans $\{1, \dots, k\}$.

On a la relation de récurrence $\delta(k, s, t) = \min(\delta(k-1, s, t), \delta(k-1, s, k) + \delta(k-1, k, t))$ et le cas simple

$$\delta(0, s, t) = \begin{cases} 0 \text{ si } s = t \\ w(s, t) \qquad \text{sinon.} \end{cases}$$

Cela donne l'algorithme suivant. L'algorithme de Floyd-Warshall s'exécute en mémoire $\mathcal{O}(|S|^2)$ et en temps $\mathcal{O}(|S|^3)$.

> Cherchez par vous même le lien entre produit de matrice (dans le sous-anneau "tropical" $(\mathbb{R}, \min, +)$) et l'algorithme de Floyd-Warshall.

[17]:
```
from copy import deepcopy
```

[18]:
```
def floyd_warshall(weight):
    """All pairs shortest paths by Floyd-Warshall

    :param weight: edge weight matrix
    :returns: weight, and True if there are negative cycles
    :complexity: :math:`O(|V|^3)`
    """
    weight = deepcopy(weight)
    V = range(len(weight))
    for k in V:  # considering paths using 0..k
        # to go from u to v
        for u in V:
            for v in V:
                weight[u][v] = min(weight[u][v],
                                   weight[u][k] + weight[k][v])
    for v in V:
        if weight[v][v] < 0:      # negative cycle found
            return weight, True
    return weight, False
```

Un exemple de graphe, comme celui utilisé dans la page Wikipédia :

[21]:
```
oo = float('+inf')

weight = [
    [0, oo, -2, oo],
    [4, 0, 3, oo],
    [oo, oo, 0, 2],
```

```
    [oo, -1, oo, 0],
]
```

[22]:
```
floyd_warshall(weight)
```

[22]:
```
([[0, -1, -2, 0], [4, 0, 2, 4], [5, 1, 0, 2], [3, -1, 1, 0]], False)
```

3.6 Résolution du problème du sac à dos par programmation dynamique

[29]:
```
def knapsack(p, v, cmax):
    """Knapsack problem: select maximum value set of items if total size not
    more than capacity

    :param p: table with size of items
    :param v: table with value of items
    :param cmax: capacity of bag
    :requires: number of items non-zero
    :returns: value optimal solution, list of item indexes in solution
    :complexity: O(n * cmax), for n = number of items
    """
    n = len(p)
    opt = [[0] * (cmax + 1) for _ in range(n + 1)]
    sel = [[False] * (cmax + 1) for _ in range(n + 1)]
    #                                --- basic case
    for cap in range(p[0], cmax + 1):
        opt[0][cap] = v[0]
        sel[0][cap] = True
    #                                --- induction case
    for i in range(1, n):
        for cap in range(cmax + 1):
            if cap >= p[i] and opt[i-1][cap - p[i]] + v[i] > opt[i-1][cap]:
                opt[i][cap] = opt[i-1][cap - p[i]] + v[i]
                sel[i][cap] = True
            else:
                opt[i][cap] = opt[i-1][cap]
                sel[i][cap] = False
    #                                --- reading solution
    cap = cmax
    solution = []
    for i in range(n-1, -1, -1):
        if sel[i][cap]:
            solution.append(i)
            cap -= p[i]
    return (opt[n - 1][cmax], solution)
```