Université de Genève

-

Sciences Informatiques

Algorithmique - TP 04

Noah Munz (19-815-489)

Novembre 2022

Contents

TP 04

1 Voyage en train (5 Points)

1.1 Fonction de récurrence

- Soit n le nombre de ville et soit $C(i,j)$ la fonction qui retourne tous les chemins possible pour aller de la ville i à la ville j. Soit donc $C'(i,j) := \min\,(C(i,j))$ la fonction qui retourne le coût optimal (minimal) pour aller de la ville i à la ville j.
  Le but de cet exercice est donc de trouver un algorithme qui déterminera $C'(i,j)$
  pour tous $i,j \in \{0\,..\,n-1\}$.

  Donnée la matrice M (triangulaire inférieure) des coûts initiaux dans l'énoncé, on cherche à remplir la matrice $T = [\ (T_{ij})_{1\leq i,j\leq n}\ ] \in \mathbb{M}_n$ des coûts optimaux où on définit T_{ij} comme suit:

$$T_{ij} := \begin{cases} \text{None} & \text{if } j > i \quad \text{(car triangulaire inférieure)} \\ C'(i,j) & \text{if } j < i \\ 0 & \text{if } i = j \end{cases}$$

  Notre relation de récurrence est donc :

$$(T_k)_{ij} = \min_{j<k<i}(\ (T_{k-1})_{ij},\quad (T_{k-1})_{kj} + (T_{k-1})_{ik}\ )$$

  (j = départ, i = arrivée). Soit v_i la ville n°i et $p(i,j)$ le chemin de v_i à v_j. Cette relation de récurrence de T tient du fait que si p est le trajet optimal pour aller de v_i à v_j (i.e. celui dont le coût vaut $C'(i,j)$) alors pour toute étape k et pour tout i,j:

  Soit $v_k \in p$ et dans ce cas $p(k,j)$ et $p(i,k)$ sont de coût optimaux
  i.e. $C'(i,j) = C'(k,j) + C'(i,k) \implies (T_k)_{ij} = C'(k,j) + C'(i,k) = (T_{k-1})_{jk} + (T_{k-1})_{ik}$

  Soit $v_k \notin p$ et dans ce cas $(T_{k-1})_{ij}$ ne change pas, i.e. $(T_k)_{ij} = (T_{k-1})_{ij} = C'(i,j)$

  Si ce n'est pas le cas, alors cela implique qu'il existe un autre trajet de coût inférieur aux 2 options plus haut ce qui contredit le fait que p soit le trajet optimal i.e. que le coût de p soit $C'(i,j)$.

  Cette relation s'exprime donc le fait que à chaque étape k on prend le minimum entre $(T_{k-1})_{ij}$ et $(T_{k-1})_{jk} + (T_{k-1})_{ik}$.

- *La formulle a-t-elle un impacte sur la manière de remplir T?*

  La formule à bien sûr un impact car c'est cette relation de récurrence qui vient déinir. En effet T est remplie selon la définition "piecewise" plus haut, et chaque $C'(i,j)$ est donc donné par la relation de récurrence.
  Il faut de plus s'assurer que les valeurs utilisés soient bien "initialisés" au moment où l'on est entrain de les remplir. Par exemple on à la relation que i doit toujours être $> j$, ou encore que $k > j$, et c'est bien la relation de récurrence qui nous donne cette structure / manière de remplire T.

  Cependant, pour ce qui est des valeurs T_{ij} en elles-mêmes, commes elles sont défini comme les minimums des $C(i,j)$ et que les minimums sont uniques, les valeurs en-elles mêmes ne vont pas changer. (Il n'y a pas différent type d'optimal soit le coût est minimum soit il ne l'est pas.)

1.3 Complexité en temps

- La complexité en temps est de $O(n^3)$, en effet on doit optimiser 3 paramètres en fonction l'une de l'autre. C'est à dire, on doit optimiser le paramètre "coût" en fonction j par rapport i et k, puis en fonction de i par rapport à j et k, puis finalement en fonction de k par rapport à i et j, on a donc 3 "depths" / niveau de profondeur.

  On aurait donc 3 boucles `for` où l'on calcule l'opération `min(a, b)` où a et b on été défini avant, ce sont des scalaires (i.e. pas des listes) `min` prend donc un temps constant (1 opération: `a > b ?`). Elle est répété au plus n^3 fois ce qui fait bien $O(n^3)$.

- La complexité de l'algorithme implémenté en 1.2 correspond bien et est bien $O(n^3)$

```
for k in range(n):
  for i in range(k+1, n):
      for j in range(min(k+1, i-1)): # ignoring those where |i-j| = 1 since there
          are no intermediary step
          if j == i: continue     # T triangular => we then iterate only over for
              the necessary values of k,i,j that is j < k < i
                                  # We could add i=k and k=j but Tij = 0 for all
                                      i==j so we dont need to cover them since
                                      they'll be 0.
          crt, candidate = T[i][j], T[i][k] + T[k][j]
          if candidate < crt : # if T_k-1(i, k)+ T_k-1(k, j) < T_k-1(i, j) we set
              T_k(i, j) to T_k-1(i, k)+ T_k-1(k, j) or else we dont change it
              T[i][j] = candidate
              P[i][j].append(k) # we add this step to the matrix of paths to keep a
                  record of the optimal intermediary step to take
```

On a bien 3 boucles `for`, celles sur j et i sont en $O(n^2)$ fois. En effet elles sont bornées par une boucle imbriqué de ce type

```
for i in range(n):
    for j in range(i):
        #<something in O(1)>
```

qui tournerait $\sum_{j=0}^{n} j = \frac{n\cdot(n+1)}{2}$ fois, et $n \cdot (n+1) = n^2 + nO(n^2)$ donc en rajoutant la boucle sur k qui fait n iterations et l'opération `min` qui est en $O(1)$ on a bien $O(n^2) \cdot O(n) \cdot O(1) = O(n^3)$

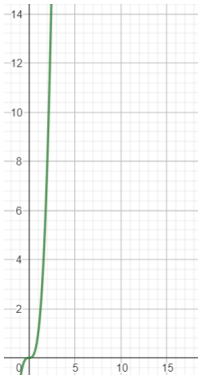

Figure 1: Fonction n^3 représentant le runtime de l'algorithme 1.2. (en y : le temps et en x la taille de la matrice en entrée.