Algorithmique (12X001) - Automne 2023

10. Programmation dynamique (2)

Enseignant: Arnaud Casteigts *Assistant: Brian Pulfer*

Nous avons vu la semaine dernière (Cours 9) deux exemples de problèmes résolus par programmation dynamique. Cette semaine, nous allons voir d'autres exemples, certains desquels ont des dépendances cycliques entre les sous-problèmes. Commençons par un bref rappel.

10.1 Rappels

Pour concevoir un programme dynamique, il faut réussir à voir le problème à résoudre comme un ensemble de sous-problèmes qui dépendent les uns des autres. Chaque sous-problème (dont le problème d'origine est un cas particulier) est résolu en s'appuyant sur la solution des sous-problèmes dont il dépend. La complexité globale dépend alors du nombre total de sous-problèmes et du coût de chacun (ce dernier dépendant à son tour du nombre de choix possibles à l'étape correspondante). Le gain est d'autant plus grand que les sous-problèmes peuvent se mutualiser, leur solution étant mémorisée pour qu'ils ne soient résolus qu'une fois chacun (principe de mémoïsation).

10.1.1 Exemple 1 : Fibonacci

On veut trouver $\text{Fib}(n)$ pour une certaine valeur de n.

- Relation entre sous-problèmes : $\text{Fib}(k) = \text{Fib}(k-1) + \text{Fib}(k-2)$,
- Nombre total de sous-problèmes : $O(n)$ ($\text{Fib}(k)$ pour tout $k \leq n$),
- Coût de chacun d'entre eux : $O(1)$ opérations (une somme entre deux entiers)

$\rightarrow$ Coût global : $O(n) \times O(1) = O(n)$.

10.1.2 Exemple 2 : Justification de texte

On veut décider où couper les lignes parmis une liste de n mots.

- Relation entre sous-problèmes : Résoudre le problème pour k mots revient à choisir le nombre l de mots d'une seule ligne, puis à résoudre le même problème pour les $k - l$ mots restants.

- Nombre total de sous-problèmes : $O(n)$ (tous les suffixes possibles de la liste initiale)
- Coût de chacun : $O(n)$ opérations ($O(n)$ choix possibles pour l, dont on sélectionne le meilleur par une opération min)

$\rightarrow$ Coût global : $O(n) \times O(n) = O(n^2)$.

10.2 Chemin le plus court

Étant donné un graphe G à n sommets, *pondéré* (les arêtes ont des coûts), et deux sommets s et t, on cherche le chemin de coût total minimum de s à t (qu'on appelera le plus court chemin). On supposera ici que les poids sont tous positifs (≥ 0) et que G est non-orienté : les arêtes fonctionnent dans les deux sens et avec le même coût ; cependant, la solution présentée ici fonctionne aussi pour les graphes orientés.

Vous connaissez peut-être déjà des algorithmes pour ce problème (Dijkstra, par exemple). Ici, nous allons tenter de le résoudre en utilisant de la programmation dynamique.

Notations : Pour tout sommet v, $N(v)$ désigne les voisins de v dans G, autrement dit tous les sommets u tels qu'une arête entre u et v existe. Le coût d'une arête uv est noté $w(u, v)$ (*weight*, en anglais).

Comment formuler le problème de manière récursive ?

Il faut réaliser la relation suivante : tout chemin de s à t arrive à t par l'un de ses voisins $v \in N(t)$. Il suffit donc de trouver les plus court chemin de s à v pour tout $v \in N(t)$, puis d'y ajouter l'arête vt. Notons $\mathcal{W}(u, v)$ le coût d'un plus court chemin d'un sommet u à un sommet v, on a la relation de récurrence suivante :

$$\mathcal{W}(s, t) = \min_{v \in N(t)}(\mathcal{W}(s, v) + w(v, t)).$$

(En cas d'égalité, on peut trancher arbitrairement.) Cette relation suggère le programme dynamique suivant :

```
coût-chemin(s,t):
   Si s=t:
     renvoyer 0
   Sinon:
     renvoyer min{coût-chemin(s,v) + w(v,t) | v ∈ N(t)}
```

Avec bien sûr de la mémoïsation pour retenir les valeurs déjà calculées.

- Nombre total de sous-problème : $O(n)$ (chemin le plus court entre s et chaque autre sommet possible)

- Coût de chacun : $O(n)$ opérations (au plus $n-1$ voisins à tester, puis on effectue un minimum parmi ce même nombre d'éléments, soit $O(n)+O(n)=O(n)$. On considère ici l'accès au poids d'une arête comme étant en temps constant.).

À première vue, on aurait donc un programme linéaire en $O(n^2)$. Mais en fait, il y a un gros problème : il ne termine pas !

10.2.1 Dépendances cycliques entre sous-problèmes

Le problème vient du fait que si notre graphe G a lui-même des cycles, alors cela induira des dépendances cycliques entre sous-problèmes : un sous-problème X est susceptible de dépendre d'un autre, qui dépend d'un autre, ..., qui dépend de X. On doit donc reformuler les sous-problèmes de manière plus subtile pour éviter cela.

Bien sûr, on peut être tenté de raisonner de manière plus globale, par exemple en interdisant de considérer des voisins déjà présent plus haut dans l'arbre de récursion. Mais alors on perd l'esprit de la programmation dynamique, qui est de ~~ne pas réfléchir~~ ne spécifier que les relations entre un problème et ses sous-problèmes directs. Comment s'en sortir ?

Idée : si un chemin jusqu'à t a une longueur k, alors la partie de ce chemin qui atteint le voisin de t doit avoir une longueur $k-1$. Notons $\mathcal{W}(u,v,k)$ le coût d'un plus court chemin de longueur $\leq k$ entre u et v. On a la nouvelle récurrence suivante :

$$\mathcal{W}(s,t,k) = \min_{v\in N(t)}(\mathcal{W}(s,v,k-1)+w(v,t)).$$

Et le programme correspondant :

```
coût-chemin(s,t,k):
   Si s=t:
     renvoyer 0
   Sinon:
     renvoyer min{coût-chemin(s,v,k-1) + w(v,t) | v ∈ N(t)}
```

que l'on appelera initialement avec la valeur $k=n-1$.

Quelle est la nouvelle complexité ?

- Nombre de sous-problèmes : $O(n^2)$ (essentiellement n choix pour t et n choix pour k).
- Coût pour chacun : $O(n)$ (comme précédemment)

On a donc résolu le problème en temps $O(n^3)$. On pourrait affiner cette valeur en tenant compte du degré maximum dans le graphe (nombre de voisins maximum), mais cela donne une idée. En l'occurrence, Dijkstra fait mieux (mais il ne peut pas s'écrire 5 lignes !).

10.2.2 Top-down versus Bottom-up

Nous avons présenté une version top-down ci-dessus, mais en fait, la version bottom-up correspond à l'algorithme très connu de Bellman-Ford, qui consiste à partir de s (plutôt que de t) et à augmenter la longueur des chemins incrémentalement.

10.2.3 Coût d'une solution versus solution elle-même ?

Vous avez peut-être remarqué que l'algorithme ci-dessus, de même que celui pour le problème de justification de texte dans le cours précédent, ne calculent pas explicitement la solution, ils se contentent d'en évaluer le coût. Il est très facile d'adapter un tel algorithme pour calculerait la solution elle-même, il suffit pour cela de mémoriser, pour chaque sous-problème X, le choix du sous-problème Y de X qui lui a apporté la meilleure solution. On stocke généralement ces informations au même endroit que pour la mémoïsation, c'est à dire dans un cache accessible globalement (table DP), ce dernier comprenant pour chaque sous-problème une valeur de coût (ou plus généralement, de qualité d'une solution) *et* un pointeur correspondant à un autre sous-problème. Une fois terminé, il suffit alors de suivre ces pointeurs pour reconstruire la solution (en temps linéaire dans la taille de la solution elle-même). Rappelons que cette mémoïsation peut être ajoutée quasi-automatiquement à n'importe quel programme récursif.

10.3 Problèmes NP-difficiles

Les problèmes que nous avons vu jusqu'à présent ont tous une complexité polynomiale en temps. Nous allons voir comment la programmation dynamique peut aussi être utilisée dans la résolution de problèmes difficiles. Bien entendu, cela ne va pas nous permettre de les résoudre en temps polynomial. Mais dans certains cas, le temps de calcul ainsi obtenu sera nettement moins mauvais qu'une approche naïve.

10.3.1 Notation O^*

Lorsqu'on s'intéresse à des complexités non polynomiales, on utilise souvent la notation O^* au lieu de O. Cette notation permet d'ignorer aussi les facteurs polynomiaux (et non plus seulement les facteurs constants et les termes dominés). Par exemple, $2^n \cdot n^3 = O^*(2^n)$. Cela permet d'identifier plus facilement les principaux obstacles à une résolution efficace des problèmes, notamment des problèmes NP-difficiles.

10.3.2 Voyageur de commerce

Pour rappel : Voyageur de commerce (TSP, pour *traveling salesperson problem* en anglais) : étant donné une ville de départ, un ensemble de n villes à visiter et un coût entre chaque paire de villes, trouver un itinéraire (une *tournée*) qui passe une fois par chaque ville et retourne au point de départ, en minimisant la somme des coûts.

Dans sa version "optimisation", le problème est NP-difficile (il est NP-complet dans sa version décisionnelle). Nous nous intéressons à la version optimisation du problème. Comme vu dans le Cours 5, une solution naïve consisterait à énumérer toutes les tournées possibles et retenir la meilleure. Il y a $n!$ tournées possibles (en fait, $(n-1)!$ si on réalise que le point de départ est quelconque, mais c'est un détail). Pour chaque tournée, on calcule sa qualité en faisant une somme de n coûts, ce qui représente un calcul négligeable à côté du goulet d'étranglement qu'est le nombre de tournées lui-même. Bref, on est en $O^*(n!)$.

Nous avions discuté l'approche gloutonne (Cours 5 également), qui est très rapide, mais qui ne trouve pas l'optimum. Voyons donc ce que la programmation dynamique peut faire pour un tel problème.

Comment décomposer le problème en sous-problèmes pertinents ? Intuitivement, on pourrait considérer tous les sous-ensembles de villes possibles et résoudre chacun comme un sous-problème distinct, en espérant que les petits sous-ensembles nous aident à résoudre les grands. Ah mais attendez... il y a 2^n sous-ensembles possibles, ça veut dire que notre complexité sera au moins de $O^*(2^n)$? Oui, en effet, mais c'est beaucoup mieux que $O^*(n!)$. Allons-y.

(Algorithme de Held-Karp.) Pour permettre plus facilement de construire une grande tournée à partir d'une plus petite, modifions le problème de sorte à finir non pas sur la ville de départ, mais sur une ville choisie. Le choix de la ville de départ n'étant pas important, on peut également la fixer, disons v_0. La forme de notre problème devient :

- $\mathrm{TSP}(S, v)$: Étant donné un ensemble de villes S (contenant v_0) et une ville d'arrivée v, trouver le coût minimum d'une tournée qui visite les villes de S en commençant par v_0 en terminant par v.

On peut maintenant mettre ces sous-problèmes en relation comme suit :

$$\mathrm{TSP}(S, v) = \min\{\mathrm{TSP}(S \setminus \{v\}, u) + w(u, v) \mid u \in S \setminus \{v, v_0\}\}$$

Cette récurrence est illustrée à la Figure 1. Le cas terminal est lorsque S ne contient plus que v_0 et une seule autre ville v, auquel cas on renvoit $w(v_0, v)$ directement.

Complexité :

- Nombre de sous-problèmes : $O(2^n \times n)$ (tous les sous-ensembles de villes, répétés pour chaque ville finale possible)

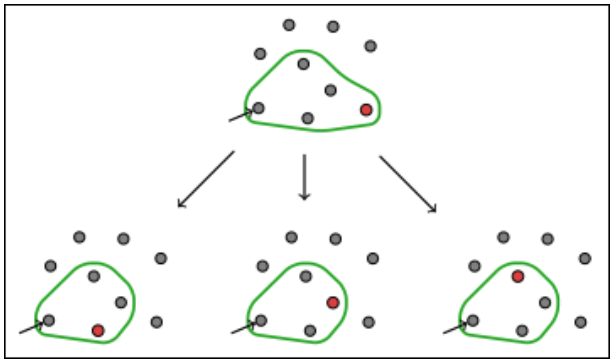

FIGURE 1 – Relation entre sous-problèmes pour le TSP

- Coût par sous-problème : $O(n)$ (appels de $\leq n$ sous-problèmes et opération minimum dessus ; il faudrait aussi examiner les opérations ensemblistes qui manipulent S, mais ces dernières semblent être effectuée en temps constant, si l'on en juge par le temps total bien connu de cet algorithme).

$\rightarrow$ Complexité totale : $O(2^n n^2) = O^*(2^n)$.

Ainsi, pour le TSP, la programmation dynamique nous permet d'obtenir une algorithme de complexité exponentielle plutôt que factorielle, ce qui est beaucoup mieux. À ce jour, il n'existe aucun algorithme plus rapide trouvant la solution optimale. (Sauf pour ordinateurs quantiques, en $O^*(1.728^n)$, là aussi avec de la programmation dynamique.)