Langages formels (11X003) - Automne 2025

4. Programmation dynamique (II)

Enseignant: Arnaud Casteigts (et F. Raynaud) *Assistant: Brian Pulfer*
Moniteurs: T. von Düring & A. Maendly

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 (en supposant les autres résolus). 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 $k \leq n$),
- Complexité 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)
- Complexité 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 pondéré G (les arêtes ont des coûts) à n sommets, 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 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 ?

$\rightarrow$ 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 choisir arbitrairement. Cette relation suggère l'algorithme :

```
cost(s,t):
   if s==t:
     return 0
   else:
     return min_{v∈N(t)}{cost(s,v) + w(v,t)}
```

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

- Nombre total de sous-problèmes : $O(n)$ (coût depuis s vers chaque autre sommet)
- Complexité de chacun : $O(n)$ opérations (au plus $n-1$ voisins à tester, puis on effectue un minimum parmi les coûts correspondants). On considère ici qu'on peut accéder en temps constant au poids d'une arête.

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

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

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 de s à t a une longueur k, alors le sous-chemin correspondant qui va de s au voisin de t a une longueur de $k-1$. 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 :

```
cost(s,t,k):
   if s==t:
     return 0
   elif k > 0:
     return min_{v∈N(t)}{cost(s,v,k-1) + w(v,t)}
   else:
     return +∞
```

On peut ensuite résoudre le problème initial en utilisant $k = n - 1$, qui est une borne supérieure sur la longueur d'un plus court chemin.

Quelle est la nouvelle complexité ?

- Nombre de sous-problèmes : $O(n^2)$ chemins potentiels depuis s (essentiellement $O(n)$ destinations et $O(n)$ longueurs possibles).
- Complexité de 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 s'écrit pas 5 lignes !).

10.2.2 Version 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 calcule pas explicitement la solution, ils se contentent d'en évaluer le coût. Il est très facile d'adapter un tel algorithme pour calculer 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 (appelée la "table DP"). Ce dernier comprend pour chaque sous-problème la valeur de coût (plus généralement, la qualité de la 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 en la taille de la solution). Rappelons que cette mémoïsation peut être ajoutée quasi-automatiquement à n'importe quel programme récursif.

Astuce de prog : En python, l'annotation `@lru_cache` active la mémoïsation. Si votre fonction ne renvoie que le coût, la table DP ne stockera que le coût. Si elle renvoie aussi la solution correspondante, la table DP stockera aussi la solution. En bref, la table DP stockera tout ce que votre fonction renvoie.

10.3 Un problème difficile

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, le voyageur de commerce (TSP) est le problème suivant : étant donné une ville de départ, un ensemble de n villes à visiter et un coût entre certaines paires de villes, trouver un itinéraire (une tournée) qui passe exactement une fois par chaque ville et retourne au point de départ, en minimisant la somme des coûts. On peut représenter une instance comme un graphe complet $G = (V, E, w)$ où V est l'ensemble des villes, E l'ensemble des arêtes entre chaque paire de ville, et où la fonction de poids $w()$ a pour valeur ∞ lorsque les deux villes ne sont pas censées être reliées.

Comme vu dans le Cours 5, une solution naïve consiste 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 son coût total en faisant une somme de n coûts, ce qui est polynomial. La complexité globale est donc essentiellement celle du nombre de tournée.

Comment décomposer le problème en sous-problèmes pertinents ?

Algorithme de Held-Karp

Intuitivement, étant donné deux sous-ensembles de villes S_1 et S_2 tels que $S_1 \subseteq S_2$, on aimerait qu'une solution pour S_1 nous aide à résoudre S_2. Le problème est que dans ce cas, on ne veut pas forcément que la ville de départ et d'arrivée soit la même, car une solution partielle n'est pas un cycle mais un chemin.

Puisque le choix de la ville de départ n'est pas important, fixons cette ville une bonne fois pour toute (disons, v_0) en ne considérant que des sous-problèmes qui contiennent v_0. Le problème devient :

- TSP(S, v) : Étant donné un ensemble de villes S (contenant v_0) et une ville d'arrivée v, trouver la meilleure tournée qui visite les villes de S depuis v_0 et termine sur v.

On peut maintenant définir les sous-problèmes naturellement : une tournée optimale pour S terminant sur v correspondant à une tournée optimale pour $S \setminus \{v\}$ terminant sur une autre ville u de S, auquel est ajouté le coût $w(u, v)$. Cela donne la récurrence suivante, aussi illustrée Figure 1 :

$$\text{TSP}(S, v) = \min_{u \in S \setminus \{v, v_0\}} \{\text{TSP}(S \setminus \{v\}, u) + w(u, v)\}$$

Le cas de base de la récurrence est lorsque S ne contient plus que v_0 et une autre ville v,

auquel cas on renvoie $w(v_0, v)$ directement. Et pour le cas particulier du problème de départ, on utilise la récurrence

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

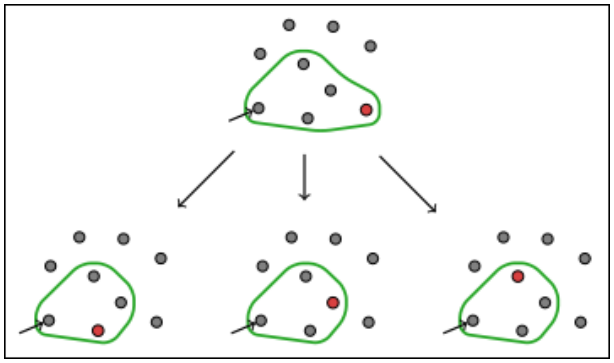

FIGURE 1 – Relation entre sous-problèmes pour le TSP

Complexité :

- Nombre de sous-problèmes : Il y a 2^{n-1} sous-ensembles de ville contenant v_0 et chacun sera considéré au plus n fois (une fois pour chaque ville finale possible), donc pour faire simple $O(2^n \cdot n)$ sous-problèmes.
- Complexité par sous-problème : $O(n)$ (appels de $\leq n$ sous-problèmes et opération minimum dessus). Il y a aussi les opérations ensemblistes qui manipulent S, mais qui peuvent être effectuées en temps constant avec les bonnes structures de données.

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

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 pour le TSP général. (Sauf pour ordinateurs quantiques, en $O^*(1.728^n)$, là aussi avec de la programmation dynamique.)