Cours n°2: Algorithmes de Programmation Dynamique pour l'Optimisation Combinatoire

Nicolas DUPIN

https://github.com/ndupin/ORteaching
http://nicolasdupin2000.wixsite.com/research

version du 19 mars 2022

Issu et étendu d'enseignements donnés à l'ENSTA Paris, Polytech' Paris-Saclay, et à l'université Paris-Saclay

Motivations

- Aujourd'hui, on va illustrer les rappels de complexité de problèmes d'optimisation et de complexité d'algorithmes.
- On avait mentionné que la programmation dynamique était un outil puissant pour prouver qu'un problème d'optimisation est polynomial, ou polynomial sur un sous -ensemble d'instances.
- On va l'illustrer sur des problèmes d'optimisation classiques et leurs variantes, mais aussi sur une application industrielle du monde réel !
- Vous avez pu voir la programmation dynamique, assez rapidement, et dans un cadre théorique général en algorithmie avancée, ce cours va illustrer la diversité des exemples et des variantes de programmation dynamique, et faire le lien aussi avec des questions de bonne implémentation.

Programmation dynamique, cadre général

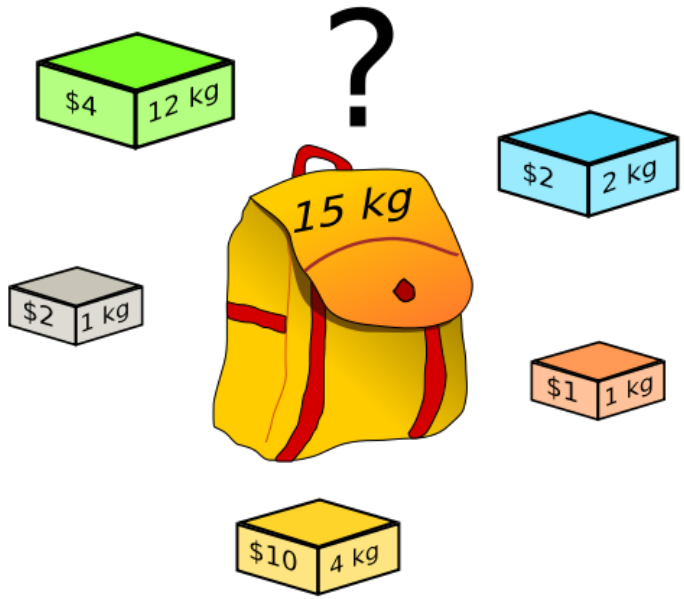
- Pour des problèmes d'optimisation, où une solution optimale est composée de bouts de solutions optimales : Principe d'optimalité de Bellman.
- ex : Plus court chemin entre deux villes, les chemins formés par les villes intermédiaire sont optimaux pour joindre ces deux villes
- Résolution s'apparente à stocker les bouts de solutions optimaux et à résoudre des relations par des formules de récurrence.
- Point clé : dans quel ordre calculer les solutions optimales partielles.
- Algorithme de Dijkstra, pour plus court chemin entre deux sommets d'un graphe (V, E) avec poids (distances) positifs, complexité polynomiale, algo en temps $O((|E| + |V|) \log |V|)$, c'est un cas particulier de la programmation dynamique !

$\Longrightarrow$ Aujourd'hui, cours/TD pour (re)découvrir et mettre en pratique ces idées sur des exemples diversifiés

Plan

Plan

Problème de sac à dos, rappel



- On dispose de N objets, de masses en kg (m_i) et de valeur financière (c_i).
- On dispose d'un sac à dos pouvant porter jusqu'à M kg.

Problème : remplir le sac à dos en maximisant la valeur financière contenue dans le sac à dos.

Un problème NP-complet (mais pas fortement NP-complet)

Et si on cherchait à énumérer toutes les possibilités ?

Algorithme brute force pour le sac à dos (1)

Brute force 1 : 2^N possibilités pour $x \in \{0,1\}^N$, on teste pour toute possibilité si la contrainte est satisfaite (en $\Theta(N)$), on évalue le coût (en $\Theta(N)$), et on compare au meilleur coût trouvé.

Algo en temps $\Theta(N.2^N)$ avec un espace mémoire additionnel en $O(1)$ (si on garde juste le coût) ou en $O(N)$ en stockant la meilleure solution courante en terme de composition.

N.B : avec la notation O^*, l'algo en $O^*(2^N)$, $O^*(a^N)$ pour dire que la complexité est en $O(poly(N)a^N)$ où $poly$ est un polynôme en N.

Eumérer les 2^N possibilités pour $x \in \{0,1\}^N$, ça peut se faire via la représentation binaire d'un entier de $[\![2^N - 1]\!]$, le k-ème bit indiquant x_k.

N.B : avec un `unsigned long` en C++,on peut aller jusqu'à $N = 64$. On peut composer `std::pair<unsigned long,unsigned long>` ou `std::vector<unsigned long>` pour avoir uen taille quelconque.

Peut on mieux faire ?

Enumération de toutes les solutions (1)

Algorithme (pseudo-code) : Enumération des solutions d'un sac à dos

Entrée : N objets définis par leur coût c_i et leur masse m_i pour tout $i \in [\![1; N]\!]$, $M > 0$ la masse totale du sac à dos

Sortie : enumKnapsack(0, [], 1, 0) affiche toutes les solutions réalisables

```
enumKnapsack(cost, list, n, m)
  if n = N + 1 : print(cost, list) end if
  if m_n + m ⩽ M :
    list = n : :list
    enumKnapsack(cost+c_n, list, n + 1, m_n + m)
    list = tl(list) // remove n
  end if
  enumKnapsack(cost, list, n + 1, m)
```

Que fait le code ci dessus, quelle complexité ?

Peut on mieux faire ?

Enumération de toutes les solutions (2)

Algo en temps $\Theta(2^N)$ quand on énumère toutes les possibilités (si on enlève condition $m_n + m \leqslant M$, remplacé par une vérification terminale) : $2^N - 1$ noeuds intermédiaire dans l'arbre binaire de recherche, le calcul du coût est amorti en $O(1)$ aux itérations. Mieux que $\Theta(N.2^N)$

On pourrait couper les calculs quand on est sûr que l'on ne peut plus ajouter d'objet par la suite : quand on est sûr de dépasser la capacité du sac à dos par la suite. L'algo précédent énumère tous les objets suivants pour tester si l'object rentre. Sur ces derniers objets , tests en $\Theta(N')$ plutôt que $\Theta(2^{N'})$ où N' est le nb d'objets restants

Soit $r_n = \min(m_n, \ldots, m_N)$ la masse minimale en prenant un objet dans $[\![n, N]\!]$.

les r_n se calculent naïvement (et indépendamment) en temps $O(N^2)$

les r_n se calculent naïvement en temps $O(N)$ avec $r_N = m_N$ et la formuler de récurrence $r_n = \min(m_n, r_{n+1})$ (cas simple de prog. dynamique)

Algorithme (pseudo-code) : Enumération des solutions d'un sac à dos

Entrée : N objets définis par leur coût c_i et leur masse m_i pour tout $i \in [\![1; N]\!]$,
$M > 0$ la masse totale du sac à dos
$r_n = \min(m_n, \ldots, m_N)$
Sortie : enumKnapsack(0, [], 1, 0) affiche toutes les solutions réalisables

```
enumKnapsack(cost, list, n, m)
  if n = N + 1 : print(cost, list) end if
  if m_n + m ⩽ M :
    list = n : :list
    enumKnapsack(cost+c_n, list, n + 1, m_n + m)
    list = tl(list) // remove n
  end if
  if r_{n+1} + m ⩽ M :
    enumKnapsack(cost, list, n + 1, m)
  else print(cost, list)
  end if
```

$\Longrightarrow$ Un calcul et stockage de r_n en temps et espace $O(N)$, évite de refaire l'équivalent de ces mêmes calculs de multiples fois (à chaque fois qu'une composition maximale de sac à dos est atteinte).

Formulation PLNE du problème de sac à dos

Remplir le sac à dos en maximisant la valeur financière contenue dans le sac à dos se formule comme le problème d'optimisation sous contrainte :

$$\begin{aligned} & \max_{x_i} \sum_{i=1}^{N} c_i \times x_i \\ s.c : \quad & \sum_{i=1}^{N} m_i \times x_i \leqslant M \\ \forall i \in [\![1, N]\!], \quad & x_i \in \{0, 1\} \end{aligned} \tag{1}$$

x_i sont des variables.
c_i, m_i et M sont des paramètres numériques

Sac à dos et programmation dynamique

- Pour le problème de sac à dos, la structure de programmation dynamique ne saute pas forcément aux yeux.
- On suppose que les masses m_i sont entières dans ce qui suit (on peut se ramener à ce cas avec des rationnels, ou quand une précision numérique est fixée)
- Variables : $(C_{i,m})_{0 \leqslant i \leqslant N, 0 \leqslant m \leqslant M}$, où $C_{i,m}$ représente le coût maximal en remplissant un sac à dos de capacité m, parmi les i premiers objets.
- $C_{N,M}$ donne la valeur du problème d'optimisation, on le calcule en calculant tous les $(C_{i,m})_{0 \leqslant i \leqslant N, 0 \leqslant m \leqslant M}$. (quelle drôle d'idée a priori ! !)

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4

i\j	00	01	02	03	04	05	06	07	08	09	10	11	12
1er objet	0												
2 premiers objets	0												
3 premiers objets	0												
4 premiers objets	0												
5 premiers objets	0												
6 premiers objets	0												
7 premiers objets	0												
8 premiers objets	0												

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4

i\j	00	01	02	03	04	05	06	07	08	09	10	11	12
1er objet	0	0	5	5	5	5	5	5	5	5	5	5	5
2 premiers objets	0												
3 premiers objets	0												
4 premiers objets	0												
5 premiers objets	0												
6 premiers objets	0												
7 premiers objets	0												
8 premiers objets	0												

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4

i\j	00	01	02	03	04	05	06	07	08	09	10	11	12
1er objet	0	0	5	5	5	5	5	5	5	5	5	5	5
2 premiers objets	0	0	5	8	8	13	13	13	13	13	13	13	13
3 premiers objets	0												
4 premiers objets	0												
5 premiers objets	0												
6 premiers objets	0												
7 premiers objets	0												
8 premiers objets	0												

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4

i\j	00	01	02	03	04	05	06	07	08	09	10	11	12
1er objet	0	0	5	5	5	5	5	5	5	5	5	5	5
2 premiers objets	0	0	5	8	8	13	13	13	13	13	13	13	13
3 premiers objets	0	0	5	8	8	14	14	19	22	22	27	27	27
4 premiers objets	0	0	6	8	11	14	14	20	22	25	28	28	33
5 premiers objets	0	0	6	8	13	14	19	21	24	27	28	33	35
6 premiers objets	0	0	6	8	13	14	19	21	24	27	30	33	36
7 premiers objets	0	0	6	10	13	16	19	23	24	29	31	34	37
8 premiers objets	0	4	6	10	14	17	20	23	27	29	33	35	38

Et après ?

On a prouvé que la valeur optimale était 38 par les calculs précédents (on le savait déjà par chance avec les algorithmes gloutons)

Ici, l'algorithme fournit l'optimum de manière générale.

Comment peut on retrouver la composition optimale du sac à dos à partir des calculs précédents, alors qu'on sait juste que la valeur optimale est 38 ?

L'affectation du sac à dos est obtenue par retour en arrière sur les disjonctions de cas dans la matrice $(C_{i,m})$

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4

i\j	00	01	02	03	04	05	06	07	08	09	10	11	12
1er objet	0	0	5	5	5	5	5	5	5	5	5	5	5
2 premiers objets	0	0	5	8	8	13	13	13	13	13	13	13	13
3 premiers objets	0	0	5	8	8	14	14	19	22	22	27	27	27
4 premiers objets	0	0	6	8	11	14	14	20	22	25	28	28	33
5 premiers objets	0	0	6	8	13	14	19	21	24	27	28	33	35
6 premiers objets	0	0	6	8	13	14	19	21	24	27	30	33	36
7 premiers objets	0	0	6	10	13	16	19	23	24	29	31	34	37
8 premiers objets	0	4	6	10	14	17	20	23	27	29	33	35	38

On prend l'objet H dans le sac à dos, $38 = 34 + 4 > 37$

Objet	A	B	C	D	E	F	G	H
Masse (kg)	2	3	5	2	4	6	3	1
Prix (euros)	5	8	14	6	13	17	10	4

i\j	00	01	02	03	04	05	06	07	08	09	10	11	12
1er objet	0	0	5	5	5	5	5	5	5	5	5	5	5
2 premiers objets	0	0	5	8	8	13	13	13	13	13	13	13	13
3 premiers objets	0	0	5	8	8	14	14	19	22	22	27	27	27
4 premiers objets	0	0	6	8	11	14	14	20	22	25	28	28	33
5 premiers objets	0	0	6	8	13	14	19	21	24	27	28	33	35
6 premiers objets	0	0	6	8	13	14	19	21	24	27	30	33	36
7 premiers objets	0	0	6	10	13	16	19	23	24	29	31	34	37
8 premiers objets	0	4	6	10	14	17	20	23	27	29	33	35	38

Composition optimale : H , G, E , D, A.

Résolution par programmation dynamique, cas général

- Variables : $(C_{i,m})_{0 \leqslant i \leqslant N, 0 \leqslant m \leqslant M}$, où $C_{i,m}$ représente le coût maximal en remplissant un sac à dos de capacité m, parmi les i premiers objets.
- Conditions limites :

$$\forall i \in [\![0, N]\!], C_{i,0} = 0$$

$$\forall m \in [\![0, M]\!], C_{0,m} = 0$$

- Formule de récurrence par disjonction de cas :

$$C_{i,m} = C_{i-1,m} \text{ si } m_i > m$$

$$C_{i,m} = \max\left\{C_{i-1,m}, C_{i-1,m-m_i} + c_i\right\} \text{ si } m_i \leqslant m$$

- En construisant les $C_{i,m}$ suivant les i croissants, on utilise dans la récurrence que des cases dont la valeur optimale a été calculée précédemment.
- $C_{N,M}$ donne la valeur du problème d'optimisation. L'affectation de sac à dos est obtenue par retour en arrière sur les disjonctions de cas dans la matrice $(C_{i,m})$

$\Longrightarrow$ Quelle Complexité ? Est-ce polynomial ?

Algorithme (pseudo-code) : Programmation dynamique, pb de sac à dos

Entrée : N objets définis par leur coût c_i et leur masse m_i pour tout $i \in [\![1; N]\!]$
Sortie : le coût de la solution optimale et la composition d'un sac à dos optimal

Initialisation :
- un vecteur de booléens $s_i = 0$ pour tout $i \in [\![0; N]\!]$
- une matrice C avec $C_{i,m} = 0$ pour tout $i \in [\![0; N]\!]$, $m \in [\![0; M]\!]$

for $i = 1$ to $N - 1$ //Construction de la matrice C
  (parallel) for $m = 1$ à M
    $C_{i,m} = C_{i-1,m}$
    if $m \geqslant m_i$ **then** $C_{i,m} = \max(C_{i,m}, C_{i-1,m-m_i} + c_i)$
  end for
end for

initialise $m = M$
for $n = N$ to 1 avec incrément $n \leftarrow n - 1$
  if $C_{n,m} \neq C_{n-1,m}$ **then** $s_n = 1$ et $m = m - m_n$
end for

retourne $C_{N,M}$ le coût optimal la solution s

Complexité de la programmation dynamique

- Construction de la matrice $C_{i,m}$: chaque case se construit en $\Theta(1)$.
- Construction de la matrice $C_{i,m}$: $\Theta(N.M)$ au total.
- Back-tracking : N opérations en $\Theta(1)$, $\Theta(N)$ au total.
- Au final, complexité en $\Theta(N.M)$ en temps et en espace mémoire.
- Est-ce polynomial ?

Complexité

- Données entrée : deux vecteurs d'entier flottants de taille N, masses et coûts, et un entier M : masse du sac à dos
- Taille des données d'entrée : $\Theta(N + \log M)$.
- Complexité non polynomiale : $N.M$ n'est pas polynomial par rapport à $(N + \log M)$.
- Ouf, le problème de sac à dos entier est NP-complet (au sens faible).
- Complexité pseudo-polynomiale : à M fixé, la programmation dynamique est polynomiale
- Le problème de sac à dos n'est pas NP-complet au sens fort.

Parallélisation de la programmation dynamique du sac à dos

Dans le pseudo code, vous avez peut être remarqué le **(parallel) for**.

Quand on a calculé les $C_{i,m}$ pour tout m, les calculs $C_{i+1,m}$ utilisent uniquement les $C_{i,m}$, ils sont indépendants, on peut les calculer en parallèle !

Parallélisation à mémoire partagée : parallélisation directe avec OpenMP (C,C++, Fortran) sur les différents threads d'une machine.

Parallélisation à mémoire distribuée : Si une machine ne peut stocker NM entiers, la parallélisation MPI (Message Passing Interface) permet la parallélisation sur des machines reliées en réseau. Rdv en M1 !

Parallélisation sur carte graphique GPU : parallélisation massive, structure de parallélisation adaptée à la programmation dynamique du problème de sac à dos standard.

Pour aller plus loin avec la parallélisation

OpenMP : parallélisation (facile) à mémoire partagée, cadre multi-threads :

`http://www.idris.fr/media/formations/openmp/idris_openmp_cours-v2.9.pdf`

`https://apps2.mdp.ac.id/perpustakaan/ebook/Karya%20Umum/Parallel_Programming_in_OpenMP.pdf`

Implémentation avec CUDA (NVIDIA) sous C++ :

`https://hal.archives-ouvertes.fr/hal-01152223/document`

`https://github.com/AJcodes/cuda_knapsack_01/`

Avec Python, numba permet de la parallélisation facile à mémoire partagée et de la parallélisation GPU :

`https://numba.readthedocs.io/en/stable/user/parallel.html`

`https://numba.readthedocs.io/en/stable/cuda/index.html`

Julia est un langage plus récent que Python, mieux conçu initialement pour la parallélisation.

Bcp d'UE en master d'informatique rattachées (obligatoires) à la filière QDCS, que vous pouvez suivre depuis les autres filières parmi les cours optionnels.

Programmmation parallèle et distribuée avec MPI : Une UE commune et obligatoire pour les filières QDCS, ANO et MPRI !

Récursivité et programmation dynamique

Les relations de récurrence peuvent être implémentées directement par récursivité :

$f(i, 0) = 0$

$f(0, m) = 0$

$f(i, m) = f(i-1, m)$ si $m_i > m$

$f(i, m) = \max\left\{f(i-1, m), f(i-1, m-m_i) + c_i\right\}$ si $m_i \leqslant m$

Quel inconvénient à cette méthode ?

Comme pour la suite de Fibonacci, on recalculerait plusieurs fois les mêmes éléments. Que faire alors ?

Memoïsation et programmation dynamique

- Idée : on applique les formules de récurrence. On commence avec une matrice de avec que des -1, sauf sur les cases terminales qu'on peut initialiser. Ensuite :
  - Si on a besoin d'une case déjà calculée, on renvoie la valeur stockée.
  - Si on a besoin d'une case non calculée (de valeur -1), on calcule la valeur correspondante par récurrence et on stocke la valeur calculée.
  - Bonus : on ne calcule que les cases de la matrice dont on a réellement besoin pour calculer la valeur terminale $C_{N,M}$
- La matrice C est indispensable de toute manière pour le backtrack et renvoyer une composition optimale du sac à dos

Sortie après implémentation séquentielle en C++

```
Dynamic Programming iterative version without parallelization  :

elapsed time: 0.00140724s
solution cost by DP: 38
print DP matrix :
0 0 5 5 5 5 5 5 5 5 5 5 5
0 0 5 8 8 13 13 13 13 13 13 13 13
0 0 5 8 8 14 14 19 22 22 27 27 27
0 0 6 8 11 14 14 20 22 25 28 28 33
0 0 6 8 13 14 19 21 24 27 28 33 35
0 0 6 8 13 14 19 21 24 27 30 33 36
0 0 6 10 13 16 19 23 24 29 31 34 37
0 4 6 10 14 17 20 23 27 29 33 35 38
knapsack composition  : 0 3 4 6 7
```

Sortie après implémentation mémoïsée en C++

```
Dynamic Programming memoized version :

solution cost by DP: 38
print DP matrix :
0 0 5 5 5 5 5 5 5 5 5 5 5
0 0 5 8 8 13 13 13 13 13 13 13 13
0 0 5 8 8 14 14 19 22 22 27 27 27
-1 0 6 8 11 14 14 20 22 25 -1 28 33
-1 -1 6 8 -1 14 19 -1 24 27 -1 33 35
-1 -1 -1 -1 -1 -1 -1 -1 24 27 -1 33 36
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 34 37
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 38
knapsack composition  : 0 3 4 6 7
```

Expérimentations de mémoïsation

Expérimentation : Combien de calculs sont évités grâce à la mémoïsation ?

ie : compter la proportion de -1 dans la matrice de programmation dynamique.

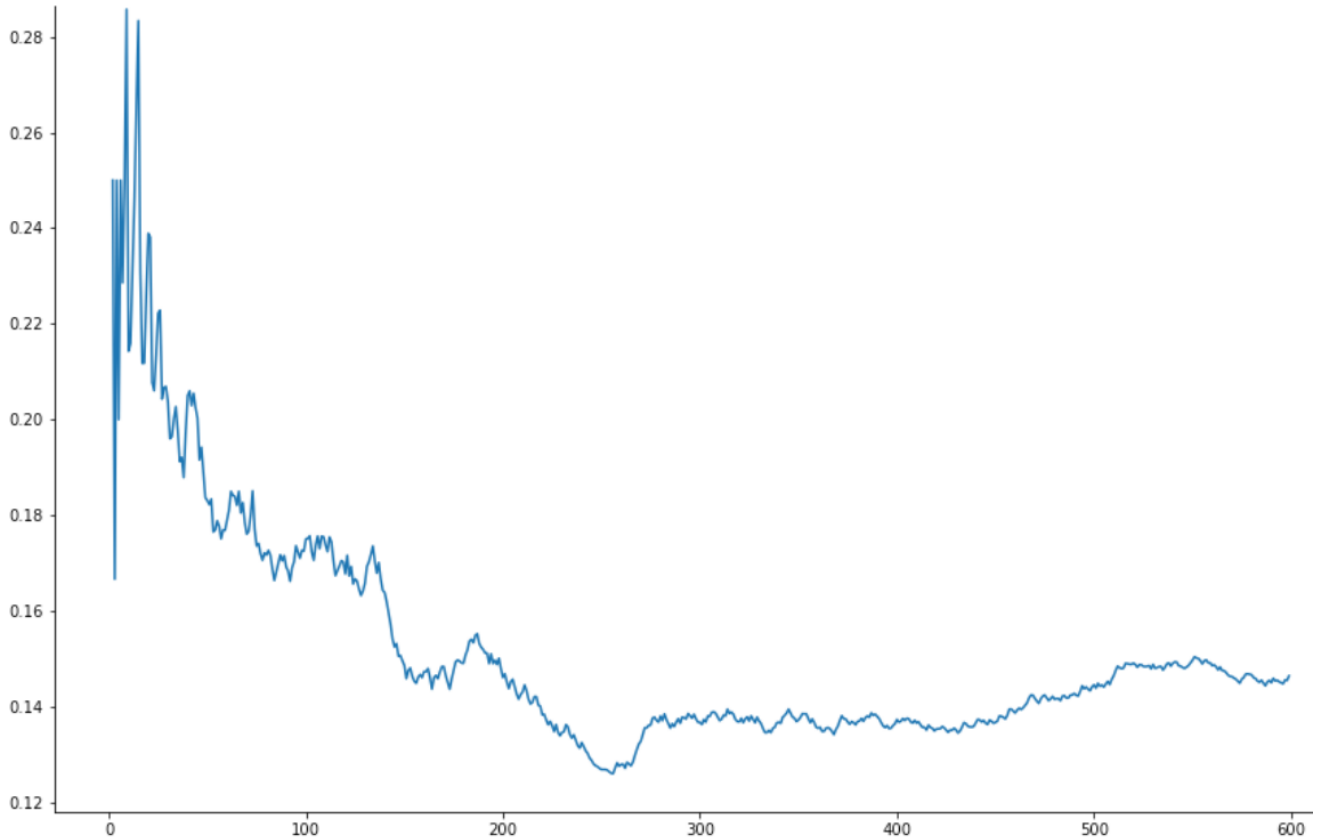
Remarque : contrairement aux algorithmes gloutons, l'ordre d'indexation des objets n'a pas d'importante.

Par quel ordre sur les objets peut on minimiser le temps de calcul, ie maximiser le nombre de -1 dans la matrice de programmation dynamique ?

Ordres possibles :
- trier les objets par masse croissante
- trier les objets par masse décroissante
- tri sur les coûts marginaux

$\Longrightarrow$ pour optimiser le nombre de cases, il vaut mieux avoir les objets les plus lourds aux indices les plus élevés

Expérience : mesure de la proportion des cases évitées par mémoïsation sur des tailles d'instances croissantes



N.B : la masse du sac à dos d'une sous instance est calculée au pro-rata de la masse des objets de la sous instance, pour que ce graphique ait un sens.

$\Longrightarrow$ Le gain est réel en pratique, mais ne correspond pas à une amélioration de la complexité

Pour aller plus loin avec la mémoïsation

La mémoïsation est une technique que vous pouvez étudier dans le cours de Programmation Fonctionnelle Avancée (Ocaml)

Dans les langages de programmation fonctionnels tels que OCaml, il est possible de réaliser la mémoïsation de manière automatique et externe à la fonction (c'est-à-dire sans modifier la fonction récursive initiale)

Mémoïsation automatique en OCaml et en Python :

`https://perso.crans.org/besson/notebooks/agreg/M%C3%A9moisation_en_Python_et_OCaml.html`

`https://www.python-course.eu/python3_memoization.php`

On peut coder de la mémoïsation automatique en C++, avec plus d'efforts :

`http://slackito.com/2011/03/17/automatic-memoization-in-cplusplus0x/`

`https://cpptruths.blogspot.com/2012/01/general-purpose-automatic-memoization.html?m=1`

Programmation dynamique et gestion du cache

- Pour l'efficacité pratique, il vaut mieux accéder à des éléments présents dans le cache plutôt que dans la RAM.
- Pour l'approche mémoïsée, il faudrait que la matrice de prog dynamqiue soit dans le cache, grosse limitation, l'espace mémoire est en $\Theta(MN)$.
- Dans le code C++, ce sera une explication à la faible performance de la version mémoïsée, la matrice $(C_{i,m})$ est trop grande, stockée en RAM.
- pour la version itérative parallélisable, on peut avoir deux tableau de taille M dans le cache : le tableau de $C_{i,m}$ en cours de calculs et les $C_{i-1,m}$ précédents. On n'écrit qu'une seule fois le résultat final dans la grande matrice stockée dans la RAM.
- pour la version itérative séquentielle, on peut avoir un seul tableau de taille M dans le cache : Le calcul de $C_{i,m}$ à $C_{i-1,m}$ se fait "in place" dans le même vecteur, en itérant de M à 0 de sorte à ce que les données réécrites ne sont plus utilisées.

$\Longrightarrow$ Questions de localité mémoire et de cache ont un fort impact pratique

Variante d'équations de programmation dynamique

- Variables : $(C_{i,m})_{0 \leqslant i \leqslant N, 0 \leqslant m \leqslant M}$, où $C_{i,m}$ représente le coût maximal en remplissant un sac à dos de capacité m, parmi les i premiers objets et i est le dernier objet pris.
- Conditions limites :

$$\forall i \in [\![1, N]\!], \forall m < m_i, C_{i,m} = -\infty$$

$$\forall m \in [\![0, M]\!], C_{0,m} = 0$$

- Formule de récurrence par disjonction de cas pour tout i, m avec $m_i \geqslant m$:

$$C_{i,m} = c_i + \max_{i' \in [\![0, i-1]\!]} C_{i',m-m_i}$$

- $\max_{i' \in [\![0,N]\!]} C_{i',M}$ donne la valeur du problème d'optimisation.
  L'affectation de sac à dos est obtenue par retour en arrière sur les disjonctions de cas dans la matrice $(C_{i,m})$

  Complexité de construction de matrice en $O(MN^2)$, moins efficace que la première version

Extensions du problème de sac à dos

1. Sac à dos multiple : le sac à dos peut porter jusqu'à M kg et a un volume maximal de V cm^3. (chaque objet i ayant un volume noté $v_i \geqslant 0$)
2. Sac à dos avec répétitions : on peut considérer prendre plusieurs fois un même objet.
3. Sac à dos à choix multiple (SCM) : I est partitionné en K sous parties J_k disjointes deux à deux, $I = \cup_k J_k$.

variante 1 Parmi les objets de J_k, on peut en prendre au plus un
variante 2 Parmi les objets de J_k, on doit exactement en prendre un.

$\Longrightarrow$ Quelle modélisation en PLNE ?

$\Longrightarrow$ Comment adapter l'algorithme de programmation dynamique ? Quelle complexité ?

Formulations PLNE

Sac à dos avec volume :

$$\begin{array}{ll} & \max_x \sum_{i=1}^{N} c_i.x_i \\ s.c : & \sum_i m_i.x_i \leqslant M \\ s.c : & \sum_i v_i.x_i \leqslant V \\ \forall i & x_i \in \{0,1\} \end{array} \quad (2)$$

Sac à dos avec répétitions :

$$\begin{array}{ll} & \max_x \sum_{i=1}^{N} c_i.x_i \\ s.c : & \sum_i m_i.x_i \leqslant M \\ \forall i & x_i \in \mathbb{N} \end{array} \quad (3)$$

Sac à dos à choix multiple (variante 1) :

$$\begin{array}{ll} & \max_x \sum_{i=1}^{N} c_i.x_i \\ s.c : & \sum_i m_i.x_i \leqslant M \\ \forall k \in [\![1;K]\!], & \sum_{i \in J_k} x_i \leqslant 1 \\ \forall i \in [\![1;N]\!] & x_i \in \{0,1\} \end{array} \quad (4)$$

Sac à dos à choix multiple (variante 2) :

$$\begin{array}{ll} & \max_x \sum_{i=1}^{N} c_i.x_i \\ s.c : & \sum_i m_i.x_i \leqslant M \\ \forall k \in [\![1;K]\!], & \sum_{i \in J_k} x_i = 1 \\ \forall i \in [\![1;N]\!] & x_i \in \{0,1\} \end{array} \quad (5)$$

Programmation dynamique et sac à dos avec volume

- Modélisation : $(C_{i,m,v})_{0 \leqslant i \leqslant N, 0 \leqslant m \leqslant M}$, où $C_{i,m,v}$ représente le coût maximal en remplissant un sac à dos de masse maximale m et de volume maximal v, parmi les i premiers objets.
- Conditions limites :

$$\forall i \in [\![0, N]\!], v \in [\![0, V]\!], \quad C_{i,0,v} = 0$$

$$\forall m \in [\![0, M]\!], v \in [\![0, V]\!], \quad C_{0,m,v} = 0$$

$$\forall i \in [\![0, N]\!], m \in [\![0, M]\!], \quad C_{i,m,0} = 0$$

- Formule de récurrence par disjonction de cas :

$$C_{i,m,v} = C_{i-1,m,v} \text{ si } m_i > m \text{ ou } v_i > v$$

$$C_{i,m,v} = \max \left\{ C_{i-1,m,v}, C_{i-1,m-m_i,v-v_i)} + c_i \right\} \text{ sinon}$$

- $C_{N,M,V}$ donne la valeur du problème d'optimisation. L'affectation de sac à dos est obtenue par retour en arrière sur les disjonctions de cas dans la matrice $(C_{i,m,v})$. Complexité temporelle et spatiale en $O(NMV)$ (le backtrack est en $O(N)$).

Sac à dos avec répétitions et prog. dynamique, v1

- Variables : $(C_{i,m})_{0 \leqslant i \leqslant N, 0 \leqslant m \leqslant M}$, où $C_{i,m}$ représente le coût maximal en remplissant un sac à dos de capacité m, parmi les i premiers objets.
- Conditions limites :

$$\forall i \in [\![0, N]\!], C_{i,0} = 0$$
$$\forall m \in [\![0, M]\!], C_{0,m} = 0$$

- Formule de récurrence par disjonction de cas :

$$C_{i,m} = \max \left\{ C_{i-1,m-k \times m_i} + k \times c_i \right\}_{k \in [\![0; m/m_i]\!]}$$

  N.B : m/m_i est ici le quotient de la division euclidienne de m par m_i

- $C_{N,M}$ donne la valeur du problème d'optimisation. L'affectation de sac à dos est obtenue par retour en arrière sur les disjonctions de cas dans la matrice $(C_{i,m})$

  Complexité temporelle en $O(NM^2)$. Peut on faire mieux ?

Sac à dos avec répétitions et prog. dynamique, v2

- Variables : $(C_{i,m})_{0 \leqslant i \leqslant N, 0 \leqslant m \leqslant M}$, où $C_{i,m}$ représente le coût maximal en remplissant un sac à dos de capacité m, parmi les i premiers objets.
- Conditions limites :

$$\forall i \in [\![0, N]\!], C_{i,0} = 0$$

$$\forall m \in [\![0, M]\!], C_{0,m} = 0$$

- Formule de récurrence par disjonction de cas :

$$C_{i,m} = C_{i-1,m} \text{ si } m_i > m$$

$$C_{i,m} = \max\{C_{i-1,m}, C_{i,m-m_i} + c_i\} \text{ si } m_i \leqslant m$$

Complexité spatiale et temporelle en $\Theta(NM)$.

N.B : ici, il faut construire une ligne $C_{i,m}$ dans l'ordre des valeurs croissantes de m pour assurer la validité, et qu'on utilise des cases optimales. Preuve de validité de l'algorithme par récurrence sur l'entier $i + m$

N.B : avec ce schéma, on perd des bonnes propriétés pour la parallélisation.

Deux variantes de sac à dos à choix multiple ?

SCM, variante 1 :

$$\begin{array}{ll} & \max_x \sum_{i=1}^{N} c_i.x_i \\ s.c : & \sum_i m_i.x_i \leqslant M \\ \forall k \in [\![1;K]\!], & \sum_{i \in J_k} x_i \leqslant 1 \\ \forall i \in [\![1;N]\!] & x_i \in \{0,1\} \end{array} \quad (6)$$

SCM, variante 2 :

$$\begin{array}{ll} & \max_x \sum_{i=1}^{N} c_i.x_i \\ s.c : & \sum_i m_i.x_i \leqslant M \\ \forall k \in [\![1;K]\!], & \sum_{i \in J_k} x_i = 1 \\ \forall i \in [\![1;N]\!] & x_i \in \{0,1\} \end{array} \quad (7)$$

Dans la suite, on ne considérera que la variante 2, la variante 1 peut s'écrire comme une instance de variante 2 :

Pour tout J_m de la variante 1, on considère un objet fictif supplémentaire, de coût et de masse nulle, cela définit J'_m.

Choisir au plus un objet dans J_m est équivalent à choisir exactement un objet dans J'_m, quitte à considérer l'objet fictif.

Transformation polynomiale, on rajoute au plus $K < N$ objets fictifs.

Programmation dynamique et SCM, indexation

$$\begin{array}{ll} & \max\limits_{x} \sum\limits_{i=1}^{N} c_i.x_i \\ s.c : & \sum_i m_i.x_i \leqslant M \\ \forall k \in [\![1;K]\!], & \sum_{i \in J_k} x_i = 1 \\ \forall i \in [\![1;N]\!] & x_i \in \{0,1\} \end{array} \tag{8}$$

Pour la programmation dynamique, on va indexer les objets dans l'ordre de sous ensembles J_j.

On aura $J_1 = \{1, \ldots, i_1\}$, $J_2 = \{1 + i_1, \ldots, i_2\}$, ..., $J_l = \{1 + i_{l-1}, \ldots, N\}$

Variables : $(C_{i,m})_{0 \leqslant i \leqslant N, 0 \leqslant m \leqslant M}$, où $C_{i,m}$ représente le coût maximal en remplissant un sac à dos de capacité m, parmi les i premiers objets et sachant que i est le dernier objet pris.

Programmation dynamique et sac à dos à choix multiple

- Variables : $(C_{i,m})_{0 \leqslant i \leqslant N, 0 \leqslant m \leqslant M}$, où $C_{i,m}$ représente le coût maximal en remplissant un sac à dos de capacité m, parmi les i premiers objets et i est le dernier objet pris.
- Conditions limites :

$$\forall i \in [\![1, N]\!], \forall m < m_i, C_{i,m} = -\infty$$

$$\forall m \in [\![0, M]\!], C_{0,m} = 0$$

- Formule de récurrence par disjonction de cas pour tout i, m avec $m_i \geqslant m$, où l est tel que $i \in J_l$:

$$C_{i,m} = c_i + \max_{i' \in [\![0, i-1]\!] \setminus J_l} C_{i',m-m_i}$$

- $\max_{i' \in J_l} C_{i',M}$ donne la valeur du problème d'optimisation. L'affectation de sac à dos est obtenue par retour en arrière sur les disjonctions de cas dans la matrice $(C_{i,m})$

  Complexité de construction de matrice en $O(MN^2)$

Programmation dynamique et sac à dos quadratique ?

Peut on généraliser au cas d'un sac à dos quadratique ?

$$\begin{aligned} & \max \sum_{i=1}^{n} c_i^{lin} x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} c_{i,j}^{quad} x_i x_j \\ s.c: \quad & \sum_{i=1}^{n} m_i x_i \leqslant M \\ \forall i \quad & x_i \in \{0,1\} \end{aligned} \tag{9}$$

Bilan partiel : sac à dos et programmation dynamique

- Un premier cas d'étude très riche !
- Construction de matrice de prog. dynamique avec objets croissants
- Backtrack pour récupérer la solution sur la matrice de prog. dynamique : la complexité du backtrack est minorée
- Implémentation récursive avec mémoïsation, très naturel.
- Bonnes propriétés de parallélisation et de gestion du cache efficace avec implémentation itérative
- Différentes règles de construction permettent d'avoir les différentes résolutions de pb de sac à dos par prog. dynamique, sauf pour le sac à dos quadratique. Complexité pseudo-polynomiale pour ces problèmes NP-complets qui ne sont pas fortement NP-complets.

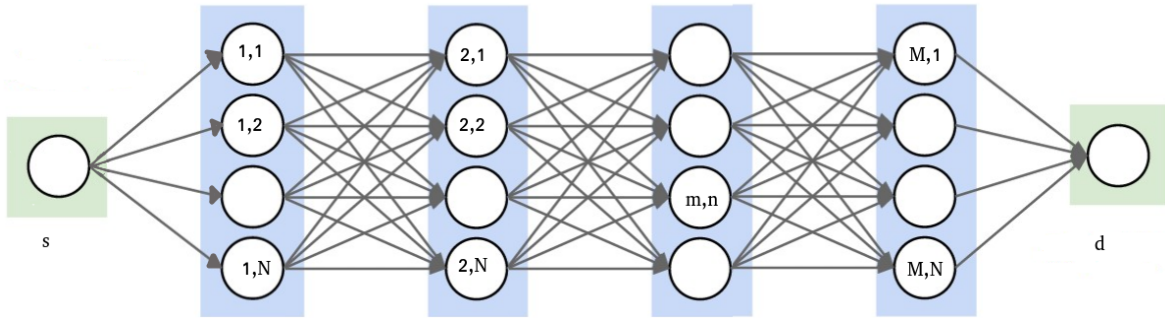
Plan

Motivation

- Soit $G = (V, E)$ un graphe pondéré, on veut calculer le plus court chemin entre un noeud source $s \in V$ et une destination $d \in V$.
- Si les distances sont toutes positives, l'algorithme de Dijkstra s'applique, c'est le cas applicatif parlant (minimiser temps/distance/prix du trajet).
- Algorithme de Dijkstra en temps $O((|E| + |V|) \log |V|)$, c'est un cas particulier de la programmation dynamique
- Les pb plus courts chemins avec des poids négatifs ont une utilité en RO, on le verra à la fin de l'UE !
- Ici, cas où il n'existe pas de cycle de poids négatif, les chemins optimaux sont alors élémentaires : ne passent au plus qu'une seule fois par chaque sommet.

Problème spécifique

Structure spécifique de graphe à M couches, chaque couche ayant N noeuds, graphe orienté sans cycle :



On note :

- $d_{m,n,n'}$ la distance entre le point m, n et le point $m+1, n'$ pour $m < M$
- a_n la distance entre l'origine et le point $1, n$
- b_n la distance entre le point M, n et la destination finale

On veut minimiser la distance d'un chemin entre la source et la destination :
- Quel algorithme de Programmation dynamique ?
- Quelle complexité ?

Résolution par programmation dynamique

- Variables : $(C_{m,n})_{1 \leqslant m \leqslant M, 0 \leqslant n \leqslant N}$, où $C_{m,n}$ représente la distance minimale entre la source et le point m, n.
- Conditions limites :

$$\forall n \in [\![1, N]\!], C_{1,n} = a_n$$

- Formule de récurrence par disjonction de cas :

$$\forall m \in [\![2, M]\!], \forall n \in [\![1, N]\!], \quad C_{m,n} = \min_{n' \in [\![1,N]\!]} (C_{n',m-1} + d_{m-1,n',n})$$

- $\min_{n \in [\![1,N]\!]}(C_{n,M} + b_n)$ donne la valeur du problème d'optimisation.

  L'affectation de sac à dos est obtenue par retour en arrière sur les disjonctions de cas dans la matrice $(C_{i,m})$

$\Longrightarrow$ Très proche dans l'esprit du sac à dos, mêmes bonnes propriétés de parallélisation et de gestion du cache

Complexité

- Données entrée : $M.N^2 + 2N$ données de distances
- Taille des données d'entrée : $\Theta(M.N^2)$.
- Complexité de la construction de la programmation dynamique : $M.N$ opérations en $O(N)$, complexité en $O(M.N^2)$.
- Complexité polynomiale, et même linéaire !
- N.B : complexité légèrement meilleure que pour l'algorithme de Dijkstra

Ajout de contrainte de ressource ?

- On associe à chaque arc la consommation d'une ressource $r_{m,n,n'}, r_n^f, r_n^l \geqslant 0$
- La ressource est limitée à une quantité $R > 0$
- ex : minimiser le temps pour aller à la pompe à essence ou recharge de véhicule électrique et ne pas tomber en panne sur le chemin.
- On peut même imaginer avoir plusieurs ressources (ex : avec des prix de péage)

Programmation dynamique avec ressource

- On suppose avoir $R, r_{m,n,n'}, r_n^f, r_n^l \in \mathbb{N}$ (s'y ramène avec des rationnels), et on réduit avec leur pgcd pour qu'ils soient les plus petits possibles
- Variables : $(C_{m,n,r})_{1 \leqslant m \leqslant M, 0 \leqslant n \leqslant N, 0 \leqslant 1 \leqslant R}$, où $C_{m,n,r}$ est la distance minimale entre la source et le point m, n en consommant moins de r ressources.
- Conditions limites :

$$\forall n \in [\![1, N]\!], \forall r \geqslant r_n^f, C_{1,n,r} = a_n$$

$$\forall n \in [\![1, N]\!], \forall r < r_n^f, C_{1,n,r} = +\infty$$

- Formule de récurrence par disjonction de cas :
  $\forall m \in [\![2, M]\!], \forall n \in [\![1, N]\!], \forall r \in [\![1, R]\!],$

$$C_{m,n,r} = \min_{n' \in [\![1,N]\!]} (C_{m-1,n',r-r_{m-1,n',n}} + d_{m-1,n',n})$$

  N.B : avec la convention $C_{m,n,r} = +\infty$ si $r \leqslant 0$.
- $\min_{n \in [\![1,N]\!]} (C_{n,M,R-r_n^l} + b_n)$ donne la valeur du problème d'optimisation.

Programmation dynamique avec ressource, complexité

- Données entrée : $M.N^2 + 2N$ données de distances et de consommations de ressource
- Taille des données d'entrée : $\Theta(M.N^2)$.
- Complexité de la construction de la programmation dynamique : MNR opérations en $O(N)$, complexité en $O(M.R.N^2)$.
- Complexité pseudo-polynomiale
- N.B : En supposant avoir $R, r_{m,n,n'}, r_n^f, r_n^l \in \mathbb{N}$ (s'y ramène avec des rationnels), et on réduit avec leur pgcd pour qu'ils soient les plus petits possibles et ne pas faire de calculs inutiles dans la programmation dynamique ou avoir un espace mémoire inutilement augmenté.

Algorithme de Floyd-Warshall(-Roy)

- On note $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un graphe quelconque sans circuit négatif.
- On note $\mathcal{V} = [\![1; V]\!]$
- Algorithme de Dijkstra et cas précédent : à partir d'une source s, on calcule tous les plus courts chemins de s à tous les autres points du graphe. Complexité temporelle $O((|E| + |V|) \log |V|)$
- Algorithme de Floyd-Warshall(-Roy) (FW) : on calcule tous les plus courts chemins entre toute paire de sommets d'un graphe, avec uniquement l'hypothèse d'absence de circuit absorbant
- Complexité de l'algorithme de FW en $O(V^3)$.
- FW peut être utilisé pour calculer juste le plus court chemin entre deux noeuds, dans un graphe avec des poids négatifs mais pas de circuit absorbant.

FW : un algorithme de programmation dynamique

- Pour tout $v, v' \in V$, on note $d_{v,v'}$ le poids (pas forcément positif) de l'arrête (v, v') si elle existe dans E, sinon $d_{v,v'} = +\infty$
- On calcule $d^v_{i,j}$ comme le plus court chemin entre $i \in \mathcal{V}$ et $j \in \mathcal{V}$ en passant par des sommets intermédiaires dans $[\![1; v]\!]$ pour tout $v \in [\![0; V]\!]$
- Conditions limites : $d^0_{i,j} = d_{i,j}$ pour tous $i, j \in \mathcal{V}$.
- Relation de récurrence :

$$\forall (i,j) \in \mathcal{V}^2, \quad \forall v \in [\![1; V]\!], \quad d^v_{i,j} = \min(d^{v-1}_{i,j}, d^{v-1}_{i,v} + d^{v-1}_{v,j}) \qquad (10)$$

- Construction des $d^v_{i,j}$ selon les valeurs croissantes de v, cela définit V itérations de V^2 calculs indépendants donc parallélisables.
- Les $d^V_{i,j}$ donnent les valeurs souhaitées, on obtient les trajets des chemins désirés par retour sur trace en utilisant (10).

Preuve de la formule d'induction

$$\forall (i,j) \in \mathcal{V}^2, \quad \forall v \in [\![1; V]\!], \quad d^v_{i,j} = \min(d^{v-1}_{i,j}, d^{v-1}_{i,v} + d^{v-1}_{v,j})$$

Preuve : Soit $(i,j) \in \mathcal{V}^2$, soit $v \in [\![1; V]\!]$. Deux cas se présentent :

• soit le plus court chemin entre i et j visitant au plus les sommets de $[\![1; v]\!]$ ne passe pas par v, et on a alors $d^v_{i,j} = d^{v-1}_{i,j}$.

• Soit ce plus court chemin passe par le sommet v. Dans ce cas, il ne peut y passer qu'une seule fois (sinon il y a un circuit, alors de valuation totale positive par hypothèse, et l'on pourrait éliminer pour construire un chemin entre i et j plus court, ce qui serait absurde). Ce plus court chemin est donc la concaténation du plus court chemin entre i et v et du plus court chemin entre v et j, parmi ceux visitant les sommets parmi les sommets de $[\![1; v-1]\!]$, et on a alors $d^v_{i,j} = d^{v-1}_{i,v} + d^{v-1}_{v,j}$.

Le résultat est acquis comme on a toujours $d^v_{i,j} \leqslant d^{v-1}_{i,j}$ et $d^v_{i,j} \leqslant d^{v-1}_{i,v} + d^{v-1}_{v,j}$.

Ouverture : que se passe t'il avec un cycle absorbant ?

Dans la preuve précédente, on voit où l'absence de circuit absorbant intervient.

En cas de circuit absorbant, avec des boucles multipliées à l'infini sur le cycle absorbant, le minimum dans $\overline{\mathbb{R}}$ est $-\infty$.

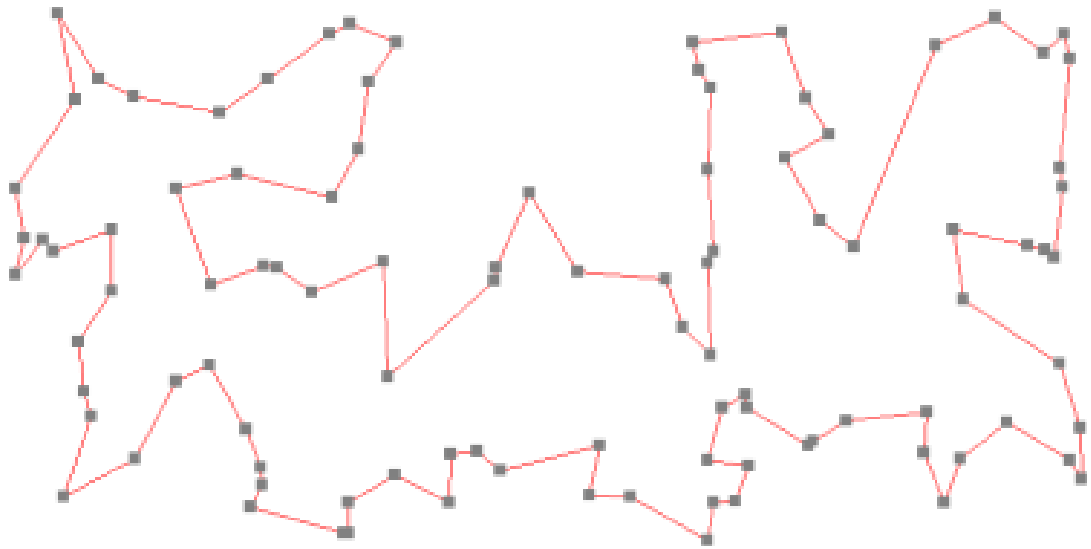
Problème de plus court chemin élémentaire : avec des distances quelconques, on peut définir le problème de plus court chemin élémentaire, ie passant au plus une fois par chacun des sommets, pb bien défini (ensemble fini de possibilités).

Le problème de plus court chemin élémentaire avec des distances quelconques est NP-complet !

Plan

Le problème de voyageur de commerce (symmétrique)

- Un voyageur de commerce doit transiter par N villes, $\mathcal{N} = [\![1; N]\!]$.
- Tous les trajets sont possibles, la distance entre les villes $n \in \mathcal{N}$ et $n' \in \mathcal{N}$ est $d_{n,n'} = d_{n',n}$ (sinon, $d_{n,n'} = +\infty$ ou un majorant).
- Minimiser la longueur du trajet du représentant de commerce.



$\Longrightarrow$ Un problème fortement NP-complet, on ne va pas avoir un algo de prog dynamique polynomial, ni même pseudo polynomial

Modélisation de programmation dynamique

On note $O_{n,E}$ pour tous $n \in [\![2; N]\!]$ et tout $E \subset [\![2; N]\!] - \{n\}$ la distance optimale de la ville 1 à la ville n en passant par toutes les villes de E exactement une fois.

Si $E = \emptyset$, $O_{n,E} = d_{1,n}$ pour tout $n \in [\![2; N]\!]$

$\forall n \in [\![2; N]\!], \forall E \subset [\![2; N]\!] - \{n\}$, $E \neq \emptyset \Longrightarrow O_{n,E} = \min\limits_{n' \in E} O_{n',E-\{n'\}} + d_{n',n}$

L'optimum du voyageur de commerce se retrouve avec $\min\limits_{n \in [\![2;N]\!]} O_{n,[\![2;N]\!]-\{n\}}$

$\Longrightarrow$ Comment calculer la matrice de programmation dynamique ?

Quel ordre de parcours ? (1)

On peut utiliser la récursivité et la mémoïsation, construit les cases nécessaires au calcul de l'optimum du problème, on ne se pose pas la question de l'ordre des calculs !

Encodage des parties de $[\![2; N]\!]$: par la représentation binaire d'un entier de 0 à $2^{N-1} - 1$, cf début du cours.

Un code pour la version récursive :

`http://liris.cnrs.fr/christine.solnon/TSPnaif.c`

$\Longrightarrow$ Quel ordre pour une implémentation séquentielle, plus efficace en C++, et pour la parallélisation ?

Quel ordre de parcours ? (2)

Dans quel ordre itérer pour garantir que les calculs qu'on utilise ont déjà été calculés à leur valeur optimale ?

Pour calculer un $O_{n,E}$, on a besoin des calculs $O_{n',E'}$ avec $n' \in E$ et $E' \subset E$

Version 1 : Un ordre (naturel ?) : calculer les $O_{n,E}$ suivant les tailles croissantes de card(E).

Pour calculer les $O_{n,E}$ avec card$(E) = k$, on a besoin uniquement de $O_{n',E'}$ avec card$(E') = k - 1$, ces calculs indépendants peuvent être parallélisés. ($N - 2$ syncronisations pour réaliser $N2^{N-2}$ calculs en tout)

$\Longrightarrow$ Question : comment énumérer les sous-ensembles de $[\![2; N]\!]$ cardinal k ? (ça se fait bien de manière itérative avec des bornes pour les k éléments)

Version 2 : On remarque que si $E' \subsetneq E$, les entiers définis par les représentations binaires vérifient $n_{E'} < n_E$.

$\Longrightarrow$ On peut itérer selon les valeurs de n_E croissantes (plus simple à coder), permet encore de bonnes parallélisations

Complexité

- Un problème fortement NP-complet, on ne va pas avoir un algo de prog dynamique polynomial, ni même pseudo polynomial
- Complexité de l'algorithme brute force : en temps $\Theta((N-1)!)$ et $\Theta(N)$ en espace mémoire additionnel.
- Nombre de cases de la matrice de prog. dynamique : $N2^{N-2}$ en espace mémoire additionnel.
- Chaque case se calcule en temps $O(N)$.
- Complexité de l'algorithme de programmation dynamique : en temps $O(N^2 2^N)$ et $\Theta(N2^N)$ en espace mémoire additionnel.

$\Longrightarrow$ Un problème fortement NP-complet, on n'a pas un algo de prog dynamique polynomial, ni même pseudo polynomial.

Plan

Voyageur de commerce avec des fenêtres de temps

- Extension du problème de voyageur de commerce : un livreur part de la ville 1 à t_1 et doit être à la ville $i \in [\![2; N]\!]$ entre t_i^{min} et t_i^{max}. (contraintes de créneaux de livraison par exemple).
- N.B : temps discrétisés, par exemple à la minute de la journée.
- On veut minimiser la longueur du trajet, tout en respectant les fenêtres de temps.
- N.B : Ici, le livreur ne s'arrête pas à chacune des villes, la livraison est rapide (ex : journaux). Des extensions existent en comptant un temps d'intervention, ex : tournée de réparation d'un technicien. Ne change pas grand chose pour l'algorithme de programmation dynamique.
- On considère un temps de parcours $t_{i,i'}$ pour transiter entre deux villes.
- Si le livreur arrive à la ville i avant t_i^{min}, il y a un temps d'attente, le livreur, ne repartira qu'à t_i^{min}.
- Le livreur ne peut pas arriver à la ville i après t_i^{max}

Voyageur de commerce avec des fenêtres de temps

- Question secondaire : a t'on une solution réalisable compte tenu des fenêtres de temps données ? Question de faisabilité, pb de décision NP-complet. Question sous-jacente
- Fonction objectif 1 : on minimise la durée de la tournée, ie l'instant de retour à 1, point de départ, ou on prouve qu'il n'y a pas de solution réalisable.
- Fonction objectif 2 : on minimise la longueur (ou le prix) du trajet de la tournée, la longueur (ou le prix) du trajet entre i et i' est toujours notée $d_{i,i'}$, ou on prouve qu'il n'y a pas de solution réalisable.
- Avec la fonction objectif 1, on peut adapter la programmation dynamique précédente. Avec la fonction objectif 2, des difficultés apparaissent, nécessitant une adaptation majeure de la prog. dynamique, algorithme d'étiquetage ("labelling algorithm")

Minimisation de la durée du trajet, prog. dynamique (1)

On note $O_{n,E}$ pour tous $n \in [\![2; N]\!]$ et tout $E \subset [\![2; N]\!] - \{n\}$ le temps minimal pour aller de la ville 1 et pouvoir repartir de n en passant par toutes les villes de E exactement une fois, et en respectant toutes les fenêtres de temps.

En cas d'impossibilité, on peut avoir la convention $O_{n,E} = +\infty$.

Initialisation (cas terminal) :

Si $E = \emptyset$, $O_{n,E} = \max(t_1 + t_{1,n}, t_n^{min})$ pour tout $n \in [\![2; N]\!]$ tq $t_1 + t_{1,n} \leqslant t_n^{max}$.

Si $E = \emptyset$, $O_{n,E} = +\infty$ pour tout $n \in [\![2; N]\!]$ tq $t_1 + t_{1,n} > t_n^{max}$.

N.B : si ce dernier cas arrive, l'instance n'est pas réalisable avec une inégalité triangulaire naturelle $t_{i,i'} \leqslant t_{i,j} + t_{j,i'}$, ici les $t_{i,i'}$ sont donnés par les plus court chemins en temps entre i, i', qui peut passer par j, $t_{i,i'} > t_{i,j} + t_{j,i'}$ est absurde dans ce contexte ...

Minimisation de la durée du trajet, prog. dynamique (2)

Equation de Bellman : soit $n \in [\![2; N]\!]$, soit $E \subset [\![2; N]\!] - \{n\}$ avec $E \neq \emptyset$.

$O_{n,E} = \min_{n' \in E'} \max(t_n^{min}, O_{n',E-\{n'\}} + t_{n',n})$

où $E' = \{n' \in E : O_{n',E-\{n'\}} + t_{n',n} \leqslant t_n^{max}\}$

N.B : si $E' = \emptyset$, on a bien $O_{n,E} = +\infty$, avec la convention usuelle.

L'optimum se retrouve avec $\min\limits_{n \in [\![2;N]\!]} O_{n,[\![2;N]\!]-\{n\}} + d_{n,1}$, la faisabilité est obtenue en n'ayant pas $+\infty$

Très analogue au problème sans fenêtres de temps. En fait, on élimine même des sous-cas, on pourrait retirer de la mémoire les cas $O_{n,E} = +\infty$, avec une structure de données bien choisie.

Implémenter suivant le cardinal croissant de E a des avantages : permet de ne pas propager les cas irréalisables. Si pour un cardinal donné, il n'y a plus de solutions réalisables, le problème est infaisable, terminaison de l'algo !

$\Longrightarrow$ Les fenêtres de temps ont un effet bénéfique pour l'algorithme de programmation dynamique ici

TSP avec fenêtres de temps (TSPTW), cas général

- Le cas général met en défaut l'algorithme de programmation dynamique.
- Sur un sous ensemble de villes, le plus court chemin en distance peut être plus long en temps qu'un autre chemin avec des temps d'attente.
- Dans la logique de prog. dynamique, on aurait éliminé le chemin plus long en distance.
- Pb : ce chemin, plus court en temps, peut être utile pour atteindre une ville où la fenêtre de temps est serrée, contrairement au chemin meilleur en coût qui ne permet pas de respecter la contrainte de fenêtres de temps
- Un chemin peut être éliminé s'il est moins bon en temps et en distance qu'un autre chemin passant par les mêmes villes, dominance plus faible.

$\Longrightarrow$ On ne doit pas stocker uniquement la meilleure distance, mais aussi à un sous ensemble donné de villes les meilleurs compromis entre distance et temps ! ! !

Algorithme d'étiquetage pour le TSPTW (1)

On construit itérativement des noeuds réalisables, $v \in V$ donnés par :

- $n_v \in [\![2; N]\!]$, la dernière ville visitée ;
- $E_v \subset [\![2; N]\!] - \{n\}$, l'ensemble des villes intermédiaires ;
- t_v : le temps minimal pour pouvoir partir de la ville n ;
- d_v : la distance cumulée ;

Initialisation :

On vérifie que pour tout n, on a $t_1 + t_{1,n} \leqslant t_n^{max}$, sinon, on s'arrête, pas de solution réalisable.

Pour tout $n \in [\![2; N]\!]$, on construit les noeuds définis par :

- $n_v = n$;
- $E_v = \emptyset$;
- $t_v = \max(t_1 + t_{1,n}, t_n^{min})$;
- $d_v = d_{1,n}$;

Algorithme d'étiquetage pour le TSPTW (2)

Propagation : On suppose avoir construit tous les noeuds réalisables et non dominés où $|E_v| = k \in [\![0, N-3]\!]$, et on va construire les tous les noeuds réalisables et non dominés où $|E_v| = k+1$.

Pour un noeud existant v donné par (n_v, E_v, t_v, d_v), on crée des nouveaux noeuds pour tout $n \in [\![2; N]\!] - (\{n_v\} \cup E_v)$ tel que $t_v + t_{n_v,n} \leqslant t_n^{max}$ avec :

- $n'_v = n$;
- $E'_v = E_v \cup \{n_v\}$;
- $t'_v = \max(t_v + t_{n_v,n}, t_n^{min})$;
- $d'_v = d_v + d_{n_v,n}$;

On élimine un noeud w' s'il existe un noeud w tel que $n_{w'} = n_w$, $E_{w'} = E_w$, $t_{w'} \geqslant t_w$ et $d_{w'} \geqslant d_w$.

Si on a M noeuds de même n_v, E_v, l'élimination peut se faire en temps $O(M \log M)$, mieux que $O(M^2)$ pour un algo naïf, ou si on avait un troisième critère de domination.

Si on n'a pas de noeuds réalisable, l'algorithme s'arrête, pas de solution réalisable

Algorithme d'étiquetage pour le TSPTW (3)

Terminaison : On suppose que l'algorithme ne s'est pas arrêté en prouvant l'infaisabilité. Soit $V' \neq \emptyset$ l'ensemble des noeuds réalisables avec $|E_v| = n - 2$.

L'optimum du problème se calcule avec $\min_{v \in V'} d_v + d_{v,1}$ (si on n'a pas de contrainte d'heure maximale au dépôt, sinon, on filtre uniquement les cas permettant de satisfaire cette dernière contrainte horaire.)

Si on veut juste calculer ce coût ou prouver la réalisabilité, on aurait pu supprimer de la mémoire les noeuds de cardinal inférieur une fois que tous les noeuds issues ont été construits.

Pour récupérer une solution optimale, on doit garder tous les noeuds générés en mémoire, et effectuer un algorithme de retour sur trace pour revenir aux sous configurations qui ont fait le chemin optimal.

Algorithme d'étiquetage, amélioration (1)

Remarque : dans la phase de propagation, si pour un noeud v, on détecte qu'une ville suivante ne peut être atteint avec la contrainte de temps, cette ville ne pourra jamais être atteinte après un noeud successeur de v.

Pour un noeud existant $v = (n_v, E_v, t_v, d_v)$, on regarde avant de propager s'il existe un $n \in [\![2; N]\!] - (\{n_v\} \cup E_v)$ tel que $t_v + d_{n_v,n} > t_n^{max}$, dans ce cas, la ville n ne pourra jamais être atteinte à temps après E_v, n_v, pas de propagation à effectuer, et on peut supprimer le noeud v !

$\Longrightarrow$ intérêt à savoir détecter rapidement une infaisabilité future au noeud courant

Algorithme d'étiquetage, amélioration (2)

On suppose ici qu'il y a une échéance pour retourner à la ville initiale 1, on doit y retourner au plus à t^{max}_{retour}. On peut essayer de vérifier si un noeud $v = (n_v, E_v, t_v, d_v)$ ne permet pas de retourner à temps à la ville initiale.

Si on a un minorant du temps restant $t^{restant}_v$ de parcours après E_v, n_v, on a une infaisabilité si $t_v + t^{restant}_v > t^{max}_{retour}$. Comment calculer un tel minorant ?

Pour toute ville $n \notin E_v \cup \{n_v\}$, on peut calculer le temps minimal pour atteindre la ville et pouvoir en repartir en partant d'une autre ville : $\min_{n' \notin E_v \cup \{n_v, n\}} \max(t_{n',n}, t_{n',n} + t^{max}_{n'} - t^{min}_{n'})$

Un minorant $t^{restant}_v$ se calcule en sommant ces minorants pour $n \notin E_v \cup \{n_v\}$, en considérant aussi le temps de retour au dépôt $\min_{n' \notin E_v \cup \{n_v, n\}} t_{n',1}$

$\rightarrow$ un tel calcul de borne est en $O((N - 3 - |E_v|))^2)$

Si on considère $\min_{n' \neq n} \max(t_{n',n}, t_{n',n} + t^{max}_{n'} - t^{min}_{n'})$, on a un minorant plus grossier, mais les calculs sont réalisés une seule fois, en temps $O(N^2)$.

$\Longrightarrow$ Compromis à trouver (expérimentalement) entre temps de calcul supplémentaire et qualité du minorant, pour éliminer plus de noeuds inutiles.

Algorithme d'étiquetage, questions d'implémentation

Le choix des conteneurs et l'accès aux données sont primordiaux pour l'efficacité de l'algorithme, à réaliser soigneusement !

Pour l'élimination par domination, on a intérêt à avoir accès facilement aux noeuds de mêmes n_v, E_v.

Ici, on n'énumère que des sous ensembles E_v réalisables, la taille des données n'est pas déterministe comme dans le cas du TSP sans contraintes de fenêtres de temps.

L'algorithme d'étiquetage se parallélise bien également, une synchronisation est nécessaire pour appliquer l'élimination par domination

ESPPRCTW : plus court chemin élémentaire avec contraintes de ressources et de fenêtres de temps

Cas similaire : on cherche le plus court chemin dans un graphe $G = (V, E)$ entre deux noeuds $s, d \in V$, avec des contraintes de fenêtres de temps pour les différents passages.

Contraintes de ressources : plusieurs ressources (indexées sur $\mathcal{R} = [\![1; R]\!]$) disponibles en quantité limitée (essence, prix péage), notée Q_r pour tout $r \in \mathcal{R}$. À chaque arrête $e = (v, v')$ est associée à une consommation de chacune des ressources, $c_{e,r}$ pour tout $r \in \mathcal{R}$.

L'élémentarité peut être vue comme des contraintes de ressources : $|V|$ ressources dans $\{0, 1\}$, budget maximal de $Q_r = 1$, indiquant si on passe par un noeud ($c_{e,v} = 1$ si l'arête e arrive à v).

Un problème qui apparaîtra à la fin de l'UE, avec des distances négatives, un problème NP-complet, fondamental à résoudre efficacement pour des méthodes de génération de colonnes !

Algorithme d'étiquetage pour ESPPRCTW (1)

Modélisation : On construit itérativement des noeuds d'un graphe correspondant à des chemins partiels réalisables. Un noeud $n \in \mathcal{N}$ est donné par :

- $v_n \in V$, le dernier sommet visitée ;
- $V_n \subset V - \{n, s\}$, l'ensemble des sommets intermédiaires ;
- t_n : le temps minimal pour pouvoir partir du sommet ville v_n ;
- d_n : la distance cumulée ;
- $q_{n,r}$: la consommation totale de la ressource r ;

Initialisation : Pour tout $v \in V_{\{}s\}$ tel que $t_s + t_{1,n} \leqslant t_v^{max}$, on construit les noeuds :

- $v_n = v$;
- $V_n = \emptyset$;
- $t_n = \max(t_s + t_{s,v}, t_v^{min})$;
- $d_n = d_{1,v}$;
- $q_{n,r} = c_{(s,v),r}$ si pour tout r $c_{(s,v),r} \leqslant Q_r$, sinon, on élimine le noeud ;

Algorithme d'étiquetage pour le ESPPRCTW (2)

Propagation : On suppose avoir construit tous les noeuds réalisables et non dominés où $|V_n| = k \in [\![0, |V| - 3]\!]$, et on va construire tous les noeuds réalisables et non dominés où $|V_n| = k + 1$.

Pour un noeud existant n donné par $(v_n, V_n, t_n, d_n, (q_{n,r}))$, on crée des nouveaux noeuds n' pour tout $v \notin \{v_n, s\} \cup V_n$ tel que $t_{v_n} + t_{v_n,v} \leqslant t_v^{max}$ et pour tout $r \in \mathcal{R}$, $c_{(v_n,v),r} + q_{n,r} \leqslant Q_r$ avec :

- $v'_n = v$;
- $V'_n = V_n \cup \{v_n\}$;
- $t'_n = \max(t_n + t_{v_n,v}, t_v^{min})$;
- $d'_n = d_n + d_{v_n,v}$;
- $q'_{n,r} = c_{(v_n,v),r} + q_{n,r}$

Domination : On élimine un noeud m' s'il existe un noeud m tel que $v_{m'} = v_m$, $V_{m'} = V_m$, $t_{m'} \geqslant t_m$, $d_{m'} \geqslant d_m$ et pour tout $r \in \mathcal{R}$, on a $q_{m',r} \geqslant q_{m,r}$. Si on a M noeuds de mêmes v_n, V_n, l'élimination se fait en temps $O(RM^2)$ (naïvement, avec toutes les comparaisons deux à deux).

Si on n'a pas de noeuds réalisables, l'algorithme s'arrête, pas de solution réalisable.

Algorithme d'étiquetage pour le ESPPRCTW (3)

Terminaison : On suppose que l'algorithme ne s'est pas arrêté en prouvant l'infaisabilité. Soit $\mathcal{N}_d \neq \emptyset$ l'ensemble des noeuds réalisables avec $v_n = d$.

L'optimum du problème se calcule avec $\min_{n \in \mathcal{N}_d} t_< n$

Si on veut juste calculer ce coût ou prouver la réalisabilité, on aurait pu supprimer de la mémoire les noeuds de cardinal inférieur une fois que tous les noeuds issues ont été construits.

Pour récupérer une solution optimale, on doit garder tous les noeuds générés en mémoire, et effectuer un algorithme de retour sur trace pour revenir aux sous configurations qui ont fait le chemin optimal.

Améliorer l'algorithme d'étiquetage du ESPPRCTW ? (1)

Avec le TSPTW, en étant obligé de passer par toutes les villes, on avait un critère d'arrêt pour faisabilité quand on sortait de la fenêtre de temps d'une ville.

Pour ESPPRCTW, on ne passe pas obligatoirement par toutes les villes, élagage non valide !

La parallélisation reste valide : en ayant construit tous les noeuds réalisables et non dominés où $|V_n| = k \in [\![0, |V| - 3]\!]$, construire tous les noeuds réalisables et non dominés où $|V_n| = k + 1$ induite des opérations indépendantes, on synchronise pour appliquer la dominance, et passer à l'itération suivante

A chaque itération, on peut avoir construit des chemins arrivant à d, ça donne des solutions réalisables, et on peut stocker la meilleure solution trouvée.

Si on a des distances positives, la distance cumulée est croissante, et on peut élaguer des noeuds dont le coût va être moins bon que la meilleure solution trouvée.

Améliorer l'algorithme d'étiquetage du ESPPRCTW ? (2)

Problème : on va avoir des ESPPRC avec des poids de distances négatifs.

Dans un graphe sans cycle absorbant, on peut calculer avec l'algorithme Floyd-Warshall en $O(|V|^3)$ la table des plus courts chemins.

Si la distance cumulée arrivant en v + la distance minimale de v à d (sans contrainte, avec FW) est moins bonne que la meilleure solution trouvée, pas la peine de propager le noeud, il ne donnera pas de solution optimale !

On peut avoir une contrainte de temps pour arriver à d, ça permet d'élaguer quand le t_n du noeud courant ne permet pas d'atteindre d avant l'échéance. On peut utiliser l'algorithme de Dijkstra ici (par positivité du temps de parcours), trouver les temps les plus rapides pour arriver à d.

Algorithmes d'étiquetage du ESPPRCTW, des références

Sadykov, R., Pessoa, A., & Uchoa, E. (2021). A bucket graph–based labeling algorithm with application to vehicle routing. Transportation Science, 55(1), 4-28.

Baldacci R, Mingozzi A, Roberti R, 2011 New route relaxation and pricing strategies for the vehicle routing problem. Operations Research 59(5) :1269–1283.

Righini, G., & Salani, M. (2006). Symmetry helps : Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints. Discrete Optimization, 3(3), 255-273.

Moungla, N. T., Létocart, L., & Nagih, A. (2010). An improving dynamic programming algorithm to solve the shortest path problem with time windows. Electronic Notes in Discrete Mathematics, 36, 931-938.

Desrochers, M., & Soumis, F. (1988). A generalized permanent labelling algorithm for the shortest path problem with time windows. INFOR : Information Systems and Operational Research, 26(3), 191-212.

$\Longrightarrow$ Un pan de recherche très riche et actuel, forts enjeux applicatifs en tournée de véhicules

Comment adapter avec des temps de stationnement ?

On peut avoir des stations rechargeant une ressource (recharge de véhicule électrique, pompe à essence)

A chaque sommet $v \in V$ est associé un temps de stationnement/rechargement, noté $t_v^{stop} \geqslant 0$, éventuellement nul comme précédemment.

Les modèle s'adaptent juste avec des modifications
$t'_n = \max(t_n + t_{v_n}^{stop} + t_{v_n,v}, t_v^{min})$.

Si on a le choix : soit on recharge et on compte un temps de rechargement, soit ce temps est économisé et on ne recharge pas ?

On peut considérer deux noeuds au lieu d'un : on peut dupliquer le sommet du graphe initial :
- sur un des noeuds, on recharge $t_{v_n}^{stop}, l_{r,v_n} > 0$
- sur l'autre $t_{v_n}^{stop}, l_{r,v_n} = 0$
- une transition nulle en trajet pour aller d'un noeud à l'autre, ou contrainte de ressource, ne visiter qu'une seul fois au plus un de ces deux noeuds.

$\Longrightarrow$ La modélisation utilise ce qui définit intrinsèquement un état présent, indépendemment du passé, et modélise des transitions possibles.

Autre adaptation : temps de livraison ou de rechargement ?

Une livraison ou un rechargement peut prendre un certain temps, négligé jusqu'ici.

A chaque sommet $v \in V$ est associé un rechargement de ressource r, noté $l_{r,v} \geqslant 0$, éventuellement nul comme précédemment.

"Difficulté" : la contrainte avec consommation cumulée dépend des rechargements éventuels, il faut additionner tous les rechargemetns précédents (on a cette information).

On peut définir plutôt $q_{n,r}$ comme la valeur restante de la ressource r, initialisée à Q_r, et dans le modèle ESPPRC avoir $q'_{n,r} = q_{n,r} - c_{(s,v),r}$ et vérifier à toute propagation $q'_{n,r} \geqslant 0$ pour ne pas tomber en panne.

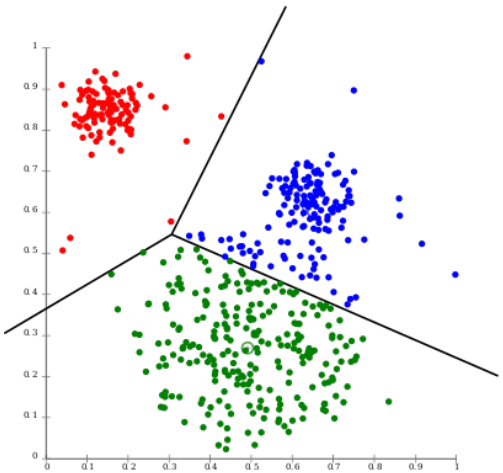
2 manières équivalentes pour ESPPRCTW, un "changement de variables" près

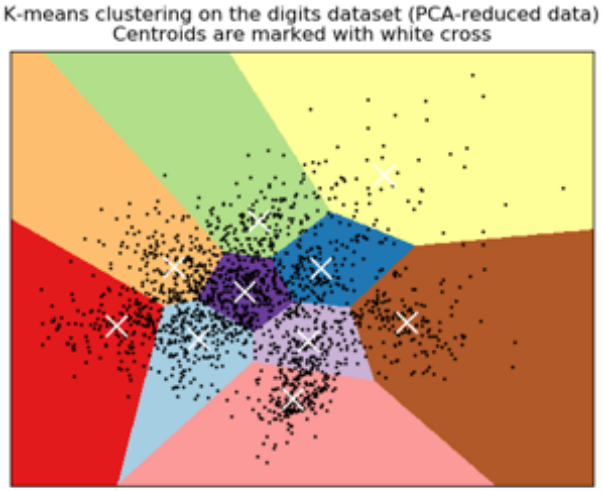
Avec des recharges, ça s'étend naturellement $q'_{n,r} = q_{n,r} - c_{(v_n,v),r} + l_{r,v_n}$

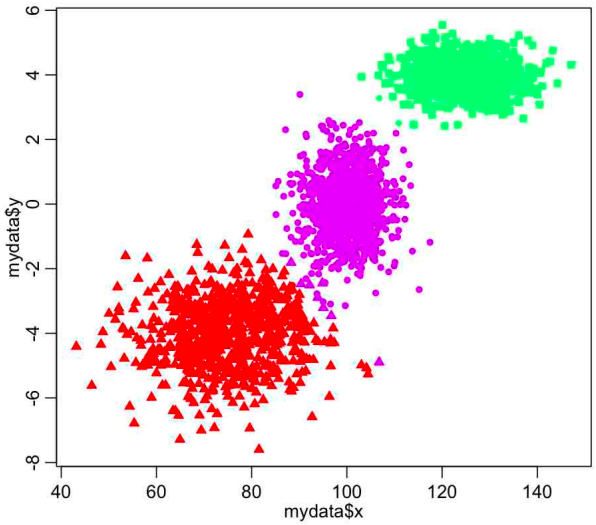
$\Longrightarrow$ La modélisation utilise préférentiellement ce qui définit intrinsèquement un état présent, indépendemment du passé, en se focalisant sur les transitions possibles.

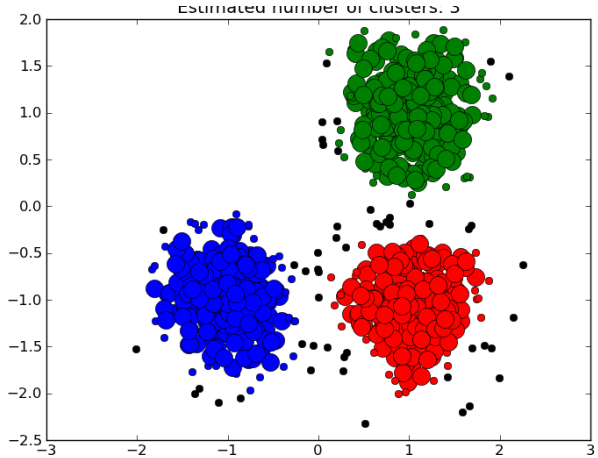
Plan

Clustering




K-means clustering on the digits dataset (PCA-reduced data)
Centroids are marked with white cross


mydata$y
mydata$x



Problèmes de clustering k-means/k-medoids

Soit $E = \{z_1, \ldots, z_N\}$ un ensemble de N éléments of $\mathbb{R}^d$. On définit $\Pi_K(E)$, l'ensemble des partitions de E en K sous-ensembles :

$$\Pi_K(E) = \left\{ P \subset \mathcal{P}(E) \;\middle|\; \forall p, p' \in P,\ p \cap p' = \emptyset \text{ and } \bigcup_{p \in P} = E \text{ and } \text{card}(P) = K \right\}$$

En définissant une fonction de coût f mesurant la dissimilarité de chaque sous ensemble de E, on définit des problèmes de clustering comme des problèmes d'optimisation indexés par $\Pi_K(E)$:

$$\min_{\pi \in \Pi_K(E)} \sum_{P \in \pi} f(P) \tag{11}$$

K-medoids : $f_{medoids}(P) = \min_{y \in P} \sum_{x \in P} \|x - y\|^2$

K-means :

$f_{means}(P) = \min_{y \in \mathbb{R}^d} \sum_{x \in P} \|x - y\|^2 = \sum_{x \in P} \left\| x - \frac{1}{\text{card}(P)} \sum_{y \in P} y \right\|^2$

K-medoids est la version discrète du problème des k-moyennes, k-means, que vous avez sans doute vu en introduction au Machine Learning.

N.B : On utilise la distance Euclidienne dans la suite, se généralise avec d'autres normes et distances. En dimension 1, cela pas bcp d'importance ...

Extensions et variantes

On peut généraliser ou considérer d'autre variantes de fonctions f

K-med, α : $f_{med}^{(\alpha)}(P) = \min_{y \in P} \sum_{x \in P} \|x - y\|^{\alpha}$

K-medoids est le cas $\alpha = 2$, K-median (discret) est le cas $\alpha = 1$. On a aussi de telles généralisations avec k-means, mais le calcul du centroïde n'est plus analytique.

Boule englobante de centre discret : $f_{ctr}^{\mathcal{D}}(P, \alpha) = \min_{y \in P} \max_{x \in P} \|x - y\|^{\alpha}$

Boule englobante avec centre quelconque : $f_{ctr}^{\mathcal{C}}(P) = \min_{y \in \mathbb{R}^2} \max_{x \in P} \|x - y\|$

Avec le centre quelconque, et $\alpha = 1$, c'est le problème "min-sum-K-radii", équivalent à "min-sum-diameter".

On peut même également considérer les problèmes de clustering min-max suivants, au lieu de min-sum :

$$\min_{\pi \in \Pi_K(E)} \max_{P \in \pi} f(P) \tag{12}$$

Application : les problèmes de K-centre (discrets et continus), sont de tels problèmes min-max, avec un calcul de boule englobante pour la fonction f.

$\Longrightarrow$ Toutes les combinaisons sont possibles, et permettent d'avoir un algo de prog dynamique et une complexité polynomiale dans le cas 1d.

Résultats de complexité pour k-medoids / k-means

On a des résultats de complexité communs à K-medoids et K-median :

- NP-difficile dans le cas general (Kariv et Hakimi 1979).
- NP-difficile dans $\mathbb{R}^2$ avec une distance Euclidienne. (Megiddo et Supowit 1984)
- Polynomiaux par programmation dynamique dans le cas 1D (on le verra dans la suite)

Des résultats de complexité assez similaires pour le pb des k-moyennes :

- NP-difficile dans le cas général, et avec une distance Euclidienne (Dasgupta 2008).
- 2-means est NP-difficile (Aloise et al 2009).
- K-means est NP-difficile en dimension 2 (Mahajan 2012).
- K-means est polynomial par programmation dynamique en dimension 1, on le verra dans la suite, algo de Wang et M. Song (2011) en temps $O(KN^2)$ et en espace mémoire $O(KN)$. Complexité améliorée par (Grønlund et al 2017) : $O(KN)$ en temps et $O(N)$ en espace mémoire.

Références

- S. Dasgupta. *The hardness of k-means clustering*, 2008.

- D. Aloise, A. Deshpande, P. Hansen, and P. Popat. *NP-hardness of Euclidean sum-of-squares clustering*. Machine learning, 75(2) :245–248, 2009.

- M. Mahajan, P. Nimbhorkar, and K. Varadarajan. *The planar k-means problem is NP-hard*. Theoretical Computer Science, 442 :13–21, 2012.

- H. Wang and M. Song. Ckmeans. *1d. dp : optimal k-means clustering in one dimension by dynamic programming*. The R journal, 3(2) :29, 2011.

- A. Grønlund et al, *Fast exact k-means, k-medians and Bregman divergence clustering in 1d*, arXiv preprint `https://arxiv.org/pdf/1701.07204.pdf`, 2017.

- O. Kariv and S. Hakimi, *An algorithmic approach to network location problems. II : The p-medians*, SIAM Journal on Applied Mathematics, vol 37, nb 3, pp 539–560, 1979.

- N. Megiddo and K. Supowit. *On the complexity of some common geometric location problems*. SIAM journal on computing, 13(1) :182–196, 1984.

- W. Hsu and G. Nemhauser, *Easy and hard bottleneck location problems*, Discrete Applied Mathematics, vol 1, nb 3, pp 209–215, 1979.

Propriété d'optimalité du clustering par intervalles

Propriété (Optimalité du clustering par intervalles en 1D)

Soient N réels, que l'on réindexe après tri : $x_1 \leqslant x_2 \leqslant \cdots \leqslant x_N$, soit $K < N$. Lorsque l'on considère les problèmes de clustering min-sum ou min-max avec les fonctions de dissimilarités précédemment définies, il existe des solutions optimales dites de clustering par intervalles, cad utilisant uniquement des clusters $\mathcal{C}_{i,i'} = \{x_j\}_{j \in [\![i,i']\!]}$

N.B : se démontre assez facilement par récurrence. Par l'absurde, on peut démontrer dans les cas min-sum que cette propriété d'optimalité est même nécessaire.

Un calcul de coût de cluster $c_{i,i'} = f(\mathcal{C}_{i,i'})$ se calcule assez facilement dans tous les cas (au pire en temps $O(N^2)$ de manière naïve pour k-medoids, en temps $O(N \log N)$ avec une recherche logarithmique).

On pourrait énumérer toutes les partitions en clusters $\mathcal{C}_{i,i'}$ et évaluer les coûts ?
Il y a $\binom{N}{K} = \dfrac{N!}{K!(N-K)!}$ telles possibilités, non polynomial en $N + \log K$.

$\Longrightarrow$ Résultat essentiel pour dériver un algorithme de prog. dynamique.

Equations de Bellman, cas min-sum clustering

On considère un problème de min-sum clustering avec fonction de dissimilarité f. Soient N réels, ré-indexés suivant le tri $x_1 \leqslant x_2 \leqslant \cdots \leqslant x_N$, soit $K < N$.

On suppose avoir calculé et stocké les $c_{i,i'} = f(\mathcal{C}_{i,i'})$ pour $i < i'$.

Pour tous $i \in [\![1, N]\!]$ et $k \in [\![1, K]\!]$, on définit $C_{i,k}$ comme le coût optimal du clustering en k sous-ensembles parmi les points indexés dans $[\![1, i]\!]$.

$C_{N,K}$ est le coût optimal que l'on cherche à calculer.

Le cas $k = 1$ est directement donné par :

$$\forall i \in [\![1, N]\!], \quad C_{i,1} = c_{1,i} \tag{13}$$

On a les relations de récurrence suivantes, en distinguant les cas sur le dernier cluster lors d'un calcul de $C_{i,k}$:

$$\forall i \in [\![1, N]\!], \ \forall k \in [\![2, K]\!], \quad C_{i,k} = \min_{j \in [\![1,i]\!]} C_{j-1,k-1} + c_{j,i} \tag{14}$$

Par valeurs croissantes de i ou k, on peut construire toutes les valeurs optimales $C_{i,k}$. On récupère une partition optimale avec un retour sur trace dans la matrice de prog. dynamique C.

Algorithm 1 : DP interval min-sum clustering for 1D instances

sort and re-index the N points by increasing values

compute cost matrix $c_{i,j}$ for all $(i,j) \in [\![1;N]\!]^2$
initialize matrix C with $C_{i,k} = 0$ for all $i \in [\![0;N]\!], k \in [\![1;K]\!]$
Set $C_{i,1} := c_{1,i}$ for all $i \in [\![1;N]\!]$

for $k = 2$ to K //Construction of the matrix C
  for (parallel) $i = 1$ to N
    set $C_{i,k} = \min_{j \in [\![1,i]\!]} C_{j-1,k-1} + c_{j,i}$
  end (parallel) for
end for

initialize $i = N$ and $\mathcal{P} = nil$ //Backtrack phase
for $k = K$ to 1 with increment $k \leftarrow k - 1$
  find $j \in [\![1,i]\!]$ such that $C_{i,k} = C_{j-1,k-1} + c_{j,i}$
  add $[\![j,i]\!]$ in $\mathcal{P}$
  $i = j - 1$
end for

return the partition $\mathcal{P}$ and the cost $C_{N,K}$

Equations de Bellman, cas min-max clustering

On considère un problème de min-max clustering avec fonction de dissimilarité f. Soient N réels, ré-indexés suivant le tri $x_1 \leqslant x_2 \leqslant \cdots \leqslant x_N$, soit $K < N$.

On suppose avoir calculé et stocké les $c_{i,i'} = f(\mathcal{C}_{i,i'})$ pour $i < i'$.

Pour tous $i \in [\![1, N]\!]$ et $k \in [\![1, K]\!]$, on définit $C_{i,k}$ comme le coût optimal du clustering en k sous-ensembles parmi les points indexés dans $[\![1, i]\!]$.

$C_{N,K}$ est le coût optimal que l'on cherche à calculer.

Le cas $k = 1$ est directement donné par : $\forall i \in [\![1, N]\!], \quad C_{i,1} = c_{1,i}$

On a les relations de récurrence suivantes, en distinguant les cas sur le dernier cluster lors d'un calcul de $C_{i,k}$:

$$\forall i \in [\![1, N]\!], \forall k \in [\![2, K]\!], \quad C_{i,k} = \min_{j \in [\![1,i]\!]} \max(C_{j-1,k-1}, c_{j,i}) \tag{15}$$

Très similaire aux cas min-sum clustering !

Une différence majeure, un calcul $C_{i,k} = \min_{j \in [\![1,i]\!]} \max(C_{j-1,k-1}, c_{j,i})$ peut se faire par recherche dichotomique, comme $c_{j,i}$ décroît avec j et $C_{j-1,k-1}$ croit avec j. De tels calculs se font en $O(\log N)$ pour le problème de K-centre en $O(KN \log N)$, crucial pour al complexité.

Problèmes de clustering polynomiaux en 1D

Théorème

Soit $E = \{x_1, \ldots, x_N\}$ un ensemble de N réels, soit $K \leqslant N$. Les différents problèmes de clustering présentés précédemment se résolvent en temps polynomial sur E. En notant α_N la complexité temporelle du calcul des coûts $f(\mathcal{C}_{i,i'})$, on a un algorithme de résolution en temps $O(\alpha_N + K.N^2) \in O(N^3 \log N)$ et espace mémoire $O(N^2)$.

Complexités obtenues en 1D :

- K-means en temps $O(N^3)$ et espace $O(N^2)$
- K-medoids et K-med(α) en temps $O(N^3 \log N)$ et espace $O(N^2)$
- Min-Sum-Diameter et Min-Sum-K-radii, variantes α en $O(KN^2)$ et espace $O(KN)$ (pas besoin de stocker les coûts)

Améliorations

Les calculs de coûts de clusters peuvent s'amortir avec des formules de récurrence, pour avoir une complexité temporelle totale en $O(N^2)$ pour k-means et $O(N^3)$ pour k-medoids.

On peut calculer une ligne de la matrice des coûts en temps $O(N)$ pour k-means et $O(N^2)$ pour k-medoids, avec une récurrence "le long d'une ligne de la matrice des coûts"

On peut "amortir" les calculs de coûts (couteux ici), avec un calcul utilisé pour toutes les cases de la matrice de prog. dynamique, et stocker en mémoire uniquement une ligne de la matrice des coûts.

Dans l'algo suivant, on évite des calcujls de cases inutiles a priori (pas d'implication sur la complexité)

Complexités obtenues en 1D :

- K-means en temps $O(KN^2)$ et espace $O(KN)$ (résultat annoncé)
- K-medoids en temps $O(N^3)$ et espace $O(KN)$

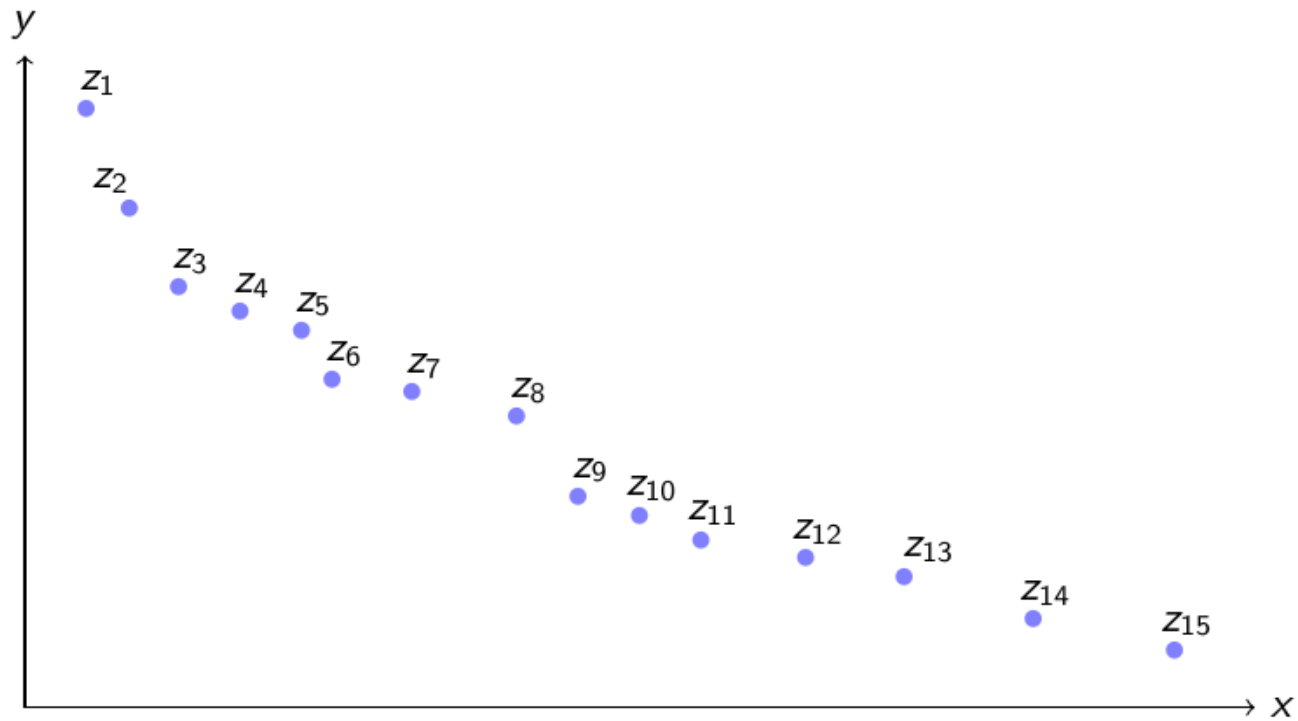
Algorithm 2 : DP k-means/k-medoids clustering for 1D instances

sort and re-index the N points by increasing values
initialize matrix M with $M_{k,i} = 0$ for all $k \in [\![1; K-1]\!], i \in [\![k; N-K+k]\!]$
store $M_{1,i} := f(\mathcal{C}_{1,i})$ for all $i \in [\![1; N-K+1]\!]$ // $O(N)$ time and space
for $i = 2$ to $N-1$
  compute and store $f(\mathcal{C}_{i',i})$ for all $i' \in [\![1; i]\!]$ // $O(N)$ time and space
  for (parallel) $k = \max(2, K+i-N)$ to $\min(K-1, i)$
    set $M_{k,i} = \min_{j \in [\![1,i]\!]} C_{k-1,j-1} + f(\mathcal{C}_{j,i})$
  end (parallel) for
  delete the stored $f(\mathcal{C}_{i',i})$ for all $i' \in [\![1; i]\!]$
end for
compute and store $f(\mathcal{C}_{i',N})$ for all $i' \in [\![1; N]\!]$ // $O(N)$ time and space
set $OPT = \min_{j \in [\![2,N]\!]} M_{K-1,j-1} + f(\mathcal{C}_{j,N})$
set $j = \text{argmin}_{j \in [\![2,N]\!]} M_{K-1,j-1} + f(\mathcal{C}_{j,N})$
delete the stored $f(\mathcal{C}_{i',N})$ for all $i' \in [\![1; N]\!]$
$i = j$
initialize $\mathcal{P} = \{[\![j; N]\!]\}$, a set of sub-intervals of $[\![1; N]\!]$.
for $k = K-1$ to 2 with increment $k \leftarrow k-1$
  compute and store $f(\mathcal{C}_{i',i})$ for all $i' \in [\![1; i]\!]$
  find $j \in [\![1, i]\!]$ such that $M_{i,k} = M_{j-1,k-1} + f(\mathcal{C}_{j,i})$
  add $[\![j, i]\!]$ in $\mathcal{P}$
  delete the stored $f(\mathcal{C}_{i',i})$ for all $i' \in [\![1; i]\!]$
  $i = j-1$
end for
return OPT the optimal cost and the partition $\mathcal{P} \cup [\![1, i]\!]$

Questions d'implémentation et de parallélisation

- On peut construire la matrice de prog. dynamique suivant les i croissants ou selon les k croissants Algorithmes 1 et 2.
- Algorithmes 1 et 2 : attention à la localité de l'espace mémoire, dans les Algos 1 et 2, il faudrait mieux inverser les indices k et i dans le stockage de la matrice au vue du parcours.
- Il vaut mieux avoir N calculs indépendants que $K < N$, Algorithme 1 a de meilleures propriétés pour la parallélisation, dont la parallélisation GPU.
- Dans l'Algorithme 2 : gestion de cache d'un vecteur de taille $O(N)$ les coûts de clusters, et un vecteur de taille $K - 2$ pour calculs courants

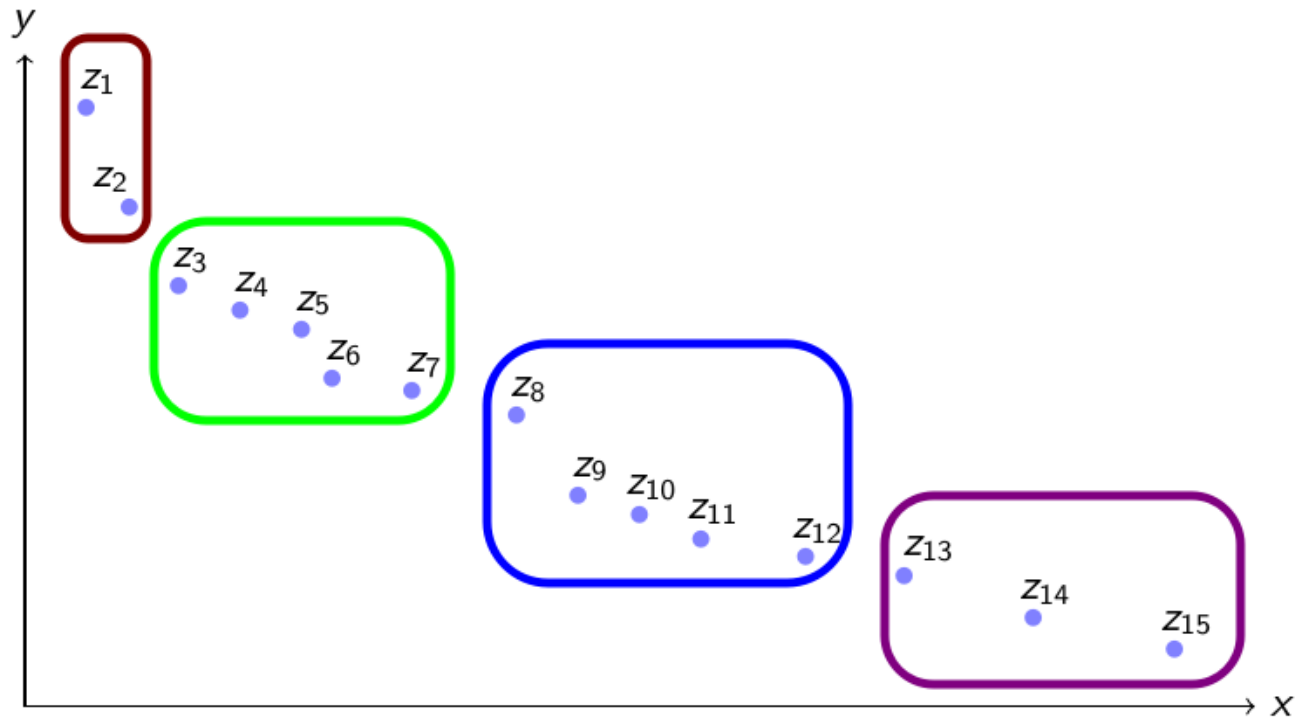
Front de Pareto discret en 2D



Un Front de Pareto (FP) correspond à un ensemble de points de $\mathbb{R}^d$ non dominés par la relation d'ordre $\preccurlyeq$:
$(a_1, \ldots, a_d) \preccurlyeq (b_1, \ldots, b_d) \iff \forall i \in \{1, \ldots, d\}, a_i \leqslant b_i$

Pour $d = 2$, ie en 2D, un FP de N points peut être indexé "de gauche à droite" en temps $O(N \log N)$.

Clustering d'un Front de Pareto discret en 2D



Intuition : sur les pb de clustering classiques, les solutions optimales seraient groupées selon l'indexation. On prouve que des solutions optimales existent sous cette forme, la réciproque peut être fausse (objectifs min-max)

$\Longrightarrow$ Base pour des algorithmes de programmation dynamique, étend le cas 1D.

Références, cas 2D PF

N. Dupin, F. Nielsen, and E. Talbi. Unified Polynomial Dynamic Programming Algorithms for P-Center Variants in a 2D Pareto Front. Mathematics, 9(4) : 453, 2021. https://doi.org/10.3390/math9040453

N. Dupin, F. Nielsen, E.G.Talbi, Clustering a 2d Pareto Front : P-center Problems Are Solvable in Polynomial Time. Optimization and Learning, In : Dorronsoro, B., Ruiz, P., de la Torre, J.C., Urda, D., Talbi, E.-G. (editors.) OLA 2020. Communications in Computer and Information Science, vol. 1173, pp. 179–191. Springer, Cham (2020). pp.179-191, 2020, Springer. https://doi.org/10.1007/978-3-030-41913-4_15

N. Dupin, F. Nielsen, E.G.Talbi, k-medoid clustering is solvable in polynomial time for a 2d Pareto front. In : Le Thi, H.A., Le, H.M., Pham Dinh, T. (eds.) WCGO 2019. Advances in Intelligent Systems and Computing, vol. 991, pp. 790–799. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-21803-4_79

N. Dupin, F. Nielsen, and E. Talbi. K-medoids and p-median clustering are solvable in polynomial time for a 2d Pareto front. ArXiv preprint. https://arxiv.org/pdf/1806.02098.pdf

Plan

Cas d'étude industriel SNCF

Version planches :

`https://f66c6e4f-6ed0-401a-ac2e-64e2f46c8140.filesusr.com/ugd/004bb2_29e6d8803c9340bbba465071ccc9d345.pdf`

Version polycopiée :

`https://www.researchgate.net/publication/323105723_Adaptation_optimisee_des_horaires_des_trains_en_presence_de_travaux`

Planches non insérées ici pour ne pas alourdir le cours avec toutes les illustrations

À retenir de l'application SNCF, côté industriel

- Une question avant l'étude, est -il possible d'optimiser les annulations de trains dans la prog dynamique, en plus des horaires ? (Avec SiOUCS, ce n'était pas possible, de même qu'avancer des horaires). Réponse : oui !
- Avec des trains circulant dans le même sens, ça ressemble à la prog. dynamique du sac à dos et variantes. Meilleure complexité en considérant des vitesses égales.
- La vitesse de parcours peut être considérée comme variable, considérer systématiquement la vitesse maximale n'est pas optimal. La prog. dynamique s'étend (plus coûteux en calcul, mais peut améliorer la fonction objectif)
- Flexibilité d'adapter la programmation dynamique à plusieurs variantes : possibilités d'annulations, écarts génériques de circulation suivant signalisation, modèles à vitesses constantes, maximales ou variables en prog. dynamique, circulation avec sens unique ou en IPCS.
- Programmation dynamique : approche efficace, implémentation directe (ici en Java, des boucles for), pas besoin de déployer de logiciels commerciaux (questions de coûts et d'audits de code externe). La version SiOUCS a été industrialisée et mis en service.

À retenir de l'application SNCF, côté prog. dynamique

- Propriété d'optimalité de non-interversion : un ingrédient essentiel pour déployer l'algo. de prog. dynamique, en progressant dans l'ordre ds trains sur planning initial.
- Avec des trains circulant dans le même sens, ça ressemble à la prog. dynamique du sac à dos standard. Avec l'IPCS, deux matrices de prog. dynamique à 3 indices (suivant le dernier sens de circulation)
- Meilleure complexité en considérant des vitesses égales dans les deux cas, même différence avec le sac à dos, définition de la matrice en prenant les i premiers trains en considérant forcément (ou non) que le train i circule.
- La programmation dynamique peut être réalisée suivant le temps croissant (forward) ou décroissant (backward).
- Recollement de prog dynamique, passage de 1 voie à 2 voies : prog. dynamique forward pour l'IPCS en 1 voie, deux prog. dynamiques backward pour la circulation en conditions normales, et recollement des sous cas optimaux en différenciant les trains circulant avant/après le passage de 1 à 2 voies.

Plan

Ce qu'il faut retenir de la Programmation Dynamique

- Algorithme qui évite des énumérations inutiles, en stockant en mémoire des solutions optimales de problèmes partiels.
- Méthode exacte associée à des problèmes d'optimisation vérifiant la propriété de Bellman, à voir comme des relations de récurrence.
- On a vu comment adapter la "gymnastique" de la programmation dynamique et des algorithmes d'étiquetage à de multiples variantes.
- Résolution en stockant en mémoire des solutions optimales de problèmes partiels. Implémentation naturelle par récursivité, avec mémoïsation. Pour une bonne parallélisation et gestion du cache, il est intéressant d'avoir une implémentation itérative.
- Complexité : un outil puissant pour démontrer des complexités de problèmes d'optimisation :
  - polynomiales (ex : plus court chemins, clustering 1D)
  - pseudo-polynomiales (sac à dos discret)
  - exponentielles également (TSP, de $\Theta(n!)$ à $\Theta(n^2 2^n)$ en temps).
- Application SNCF : approche efficace, implémentation directe, pas besoin de déployer de logiciels à licence commerciale (coûts et audits de code).

Pourquoi le terme de Programmation Dynamique ?

- "Dynamique" = aspect de progression temporelle. Equation de Bellman selon progression temporelle, le futur est déterminé par la connaissance parfaite de la condition initiale.
- Application ordonnancement SNCF : une indexation temporelle est bien présente. Mais les autre cas ?
- Progression selon un axe à définir pour calculer une séquence de sous-problèmes optimaux :
  - Clustering : structure 1D (ou 2D PF) donne un ordre de parcours pour la prog. dynamique, contrairement aux dimensions supérieures
  - Sac à dos : ordre de parcours des objets, tous les ordres sont possibles pour avoir un algo de prog dynamique, mais il faut bien en choisir un.
  - TSP : ordre de parcours des sous parties, selon le cardinal croissant ou l'ordre induit par la représentation binaire en entier.
  - Floyd-Warshall-Roy : ensemble des sommets intermédiaires croissants au sens de l'inclusion.

Pour la suite

- La suite du cours se focalisera sur la modélisation et résolution générale de PLNE.
- La programmation dynamique sera implémentée pour le pb de sac à dos en C++, version mémoïsée et itérative.
- Voyageur de commerce : la prog. dynamique n'est pas l'approche exacte la plus efficace en pratique (en temps de calculs, la meilleure complexité de pire cas est bien fournie par la prog. dynamique)
- La programmation dynamique servira pour des méthodes de programmation mathématique avancée (génération de coupes et/ou de colonnes).
- Méthodes de décomposition (exactes et heuristiques) : on résoudra des plus petits sous-problèmes, important de détecter qu'un sous problème se résout efficacement (ie complexité polynomiale voire pseudo-polynomiale).