Evaluation des algorithmes de ranking pour classer des produits sur un site de e-commerce

Flavie Bertrand et Marion Tremblay

2025-12-08

1. Description du problème et objectif

1.1. Problème général de ranking

On considère :

- $i \in \{1, 2, \ldots, n\}$: l'élément i
- un score associé à chaque élément : $s_i \in \mathbb{R}$
- une variable binaire $x_{i,p} \in \{0, 1\}$ qui vaut 1 si l'élément i est placé à la position p (0 sinon)
- un poids w_p associé à la position p

L'objectif est de déterminer un classement des k premiers éléments maximisant le score total.

$$\max_x \sum_{i=1}^{n} \sum_{p=1}^{k} w_p \, s_i \, x_{i,p}$$

Sous contraintes :

- Un élément ne peut occuper au plus qu'une seule position :

$$\sum_{p=1}^{k} x_{i,p} \leq 1, \quad \forall i$$

- Chaque position doit être occupée par exactement un élément :

$$\sum_{i=1}^{m} x_{i,p} = 1, \quad \forall p$$

- Contraintes de groupes :

On définit des groupes :

$$G_g \subseteq \{1, \ldots, m\}, \quad g = 1, \ldots, G$$

On impose que chaque groupe G_g apparaisse au plus α_g fois dans les k premiers :

$$\sum_{i \in G_g} \sum_{p=1}^{k} x_{i,p} \leq \alpha_g, \quad \forall g$$

1.2. Exemple concret

1.2.1. Contexte

On applique ce problème pour sélectionner les 10 produits les plus pertinents pour les afficher sur la page d'accuei d'un site de e-commerce. Chaque produit a un score s_i correspondant à la moyenne des notes données par les utilisateurs. On souhaite maximiser ce score tout en respectant certaines contraintes : - Diversité des catégories : pas plus de 3 produits de la même catégorie - Limiter la domination d'une marque : pas plus de 2 produits de la même marque - Contrainte marketing : au moins un produit sponsorisé doit apparaître dans le top 10

Pour répondre à ce problème on utilise la base de données …

1.2.2. Formulation mathématique

La fonction objectif est :

$$\max \sum_{k=1}^{10} w_k \, s_{\pi(k)}$$

où :

- $\pi(k)$ est le produit placé à la position k,
- w_k est le poids associé à la position k.

Sous contraintes

- Chaque position contient exactement un produit :

$$\sum_{i=1}^{n} x_i^k = 1 \quad \forall k$$

- Un produit ne peut être affiché qu'une seule fois :

$$\sum_{k=1}^{10} x_i^k \leq 1 \quad \forall i$$

- Maximum 3 produits par catégorie :

$$\sum_{i \in \text{cat } c} \sum_{k=1}^{10} x_i^k \leq 3$$

- Maximum 2 produits par marque :

$$\sum_{i \in \text{brand } c} \sum_{k=1}^{10} x_i^k \leq 2$$

- Au moins 1 produit sponsorisé dans le top 10 :

$$\sum_{i \in \text{sponsor}} \sum_{k=1}^{10} x_i^k \geq 1$$

Cette dernière contrainte est équivalente à avoir maximum 9 produits non sponsorisé dans le top 10, soit :

$$\sum_{i \in \text{no sponsor}} \sum_{k=1}^{10} x_i^k \leq 9$$

2. Approche heuristique

3. Solution 1 : programmation dynamique

4. Solution 2 : programmation dynamique et tas

5. Comparaison des résultats

6. Complexité des algorithmes (par le calcul)

Voici nos 4 algorithmes :

1. **Algorithme naïf R**
2. **Algorithme dynamique R**
3. **Algorithme dynamique C++**
4. **Algorithme dynamique C++ amélioré (beam search)**

Pour chacun, nous analysons la complexité **temps** (meilleur, pire, moyenne) Les paramètres en jeu sont : - **n** : nombre d'items (taille de l'entrée) - **G** : nombre de groupes - **k** : nombre de positions à remplir - **S** : nombre d'états effectivement atteints dans la DP - **M** : nombre théorique maximal d'états = $\prod_{g=1}^{G}(\text{max\_cap}_g + 1)$ - $\bar{g}$: nombre moyen de groupes par item

1. Algorithme Naïf Glouton (R)

Sélection gloutonne : pour chaque position p, parcourir tous les candidats restants et choisir celui au meilleur gain immédiat $w_p \times \text{score}_i$.

Analyse détaillée

Étape 1 - Initialisation : $O(G)$

- Création des structures de données
- Négligeable devant la boucle principale

Étape 2 - Boucle principale

Pour chaque position $p = 1, \ldots, k$:

- Nombre d'items restants à examiner : $n - (p - 1)$
- Pour chaque item candidat :
  - Parser les groupes : $O(\bar{g})$
  - Vérifier les contraintes : $O(\bar{g})$ comparaisons
  - Calculer le gain : $O(1)$
- Mise à jour des compteurs : $O(\bar{g})$

Coût pour la position p :

$$T_p = O((n - p + 1) \times \bar{g})$$

Coût total de la boucle :

$$T_{\text{boucle}} = \sum_{p=1}^{k} O((n - p + 1) \times \bar{g}) = O\left(\bar{g} \times \left(kn - \frac{k(k-1)}{2}\right)\right) \approx O(kn\bar{g})$$

Complexité temporelle

$$\boxed{T_{\text{naïf}}(n) = O(kn\bar{g})}$$

Meilleur cas : $\boxed{T(n) = O(kn)}$ si $\bar{g} = O(1)$

Cas moyen : $\boxed{T(n) = O(kn\bar{g})}$

Pire cas : $\boxed{T(n) = O(knG)}$ si $\bar{g} = G$

2. Algorithme DP (R)

Programmation dynamique avec états définis par $(p, c_1, \ldots, c_G)$ où c_g = nombre d'items du groupe g déjà utilisés.

Analyse détaillée

Étape 1 - Prétraitement : $O(nG)$

- Parser les groupes de chaque item : $O(n\bar{g})$
- Créer la matrice binaire `item_groups` : $O(nG)$

Étape 2 - Programmation dynamique

Structure : `DP[[p+1]][[key]]` avec `key = "c1,c2,...,cG"`

Pour chaque position $p = 1, \ldots, k$:

- États actifs au niveau $p-1$: S_p états
- Pour chaque état actif :
  - Parser la clé : $O(G)$
  - Pour chaque item $i = 1, \ldots, n$:
    - Vérifier si utilisé : $O(p)$
    - Calculer nouveaux compteurs : $O(G)$
    - Vérifier contraintes : $O(G)$
    - Créer nouvelle clé : $O(G)$
    - Mise à jour DP : $O(1)$

Coût pour un état et un item : $O(p+G)$

Coût pour la position p : $T_p = O(S_p \times n \times (p+G))$

En supposant $S_p \approx S$ constant et $G \geq k$:

$$T_{\text{DP}} = \sum_{p=1}^{k} O(S \times n \times G) = O(SnkG)$$

Étape 3 - Recherche du meilleur : $O(S \times G)$

Complexité temporelle

$$\boxed{T_{\text{DP-R}}(n) = O(SnkG)}$$

Meilleur cas : $\boxed{T(n) = O(nkG)}$ si $S = O(1)$

Cas moyen : $\boxed{T(n) = O(SnkG)}$ avec $S \ll M$

Pire cas : $\boxed{T(n) = O(MnkG)}$ où $M = \prod_{g=1}^{G}(\text{max\_cap}_g + 1)$ (exponentiel en G)

3. Algorithme DP (C++)

Même logique que DP R, mais avec `unordered_map` et optimisations C++.

Analyse détaillée

Prétraitement : $O(nG)$

- Parsing avec `stringstream` : $O(n\bar{g})$
- Construction matrice : $O(nG)$

Programmation dynamique

Pour chaque position p, chaque état S_p, chaque item n :

- Parse key : $O(G)$
- Vérifier si utilisé (std::find) : $O(p)$
- Calculer compteurs : $O(G)$
- Make key : $O(G)$
- Hash + lookup : $O(G)$ pour le hashing, $O(1)$ amorti pour l'accès
- Insert/update : $O(1)$ amorti

Coût par transition : $O(p + G)$

Si $G \geq k$: $T_{\text{DP}} = O(SnkG)$

Complexité temporelle

$$\boxed{T_{\text{DP-C++}}(n) = O(SnkG)}$$

Meilleur cas : $\boxed{T(n) = O(nkG)}$ si $S = O(1)$

Cas moyen : $\boxed{T(n) = O(SnkG)}$

Pire cas : $\boxed{T(n) = O(MnkG)}$ avec M exponentiel en G

Note : Même complexité asymptotique que DP R, mais facteur constant 5-20$\times$ plus petit en pratique.

4. Algorithme dynamique amélioré (C++)

DP avec pruning : conserve uniquement les `beam_size` meilleurs états à chaque niveau.

Analyse détaillée

Soit $B = \min(\text{beam\_size}, M)$ le nombre effectif d'états conservés.

Étape 1 - Tri initial : $O(n \log n)$

Items triés par score décroissant pour améliorer la qualité des états gardés.

Étape 2 - DP avec tas (priority_queue)

Pour chaque position p, chaque état (max B), chaque item n :

- Extraction des états du tas : $O(B \log B)$ (une fois par niveau)
- Pour chaque état $\times$ item :
  - Calcul compteurs : $O(G)$
  - Make key : $O(G)$
  - Push dans tas : $O(\log B)$
  - Pruning si nécessaire : $O(\log B)$

Coût par niveau : $T_p = O(B \log B) + O(nB(G + \log B))$

Si $G \geq \log B$ (généralement vrai) :

$$T_p = O(nBG)$$

Sur k niveaux :

$$T_{\text{DP}} = O(nkBG)$$

Étape 3 - Recherche meilleur : $O(kB)$

Étape 4 - Reconstruction : $O(n)$

Complexité temporelle

$$\boxed{T_{\text{Beam}}(n) = O(n \log n + nkBG)}$$

Si $nkBG \gg n \log n$: $\boxed{T_{\text{Beam}}(n) = O(nkBG)}$

Meilleur cas : $\boxed{T(n) = O(n \log n + nkG)}$ si $B = 1$

Cas moyen : $\boxed{T(n) = O(n \log n + nkBG)}$ avec B modéré (100-10000)

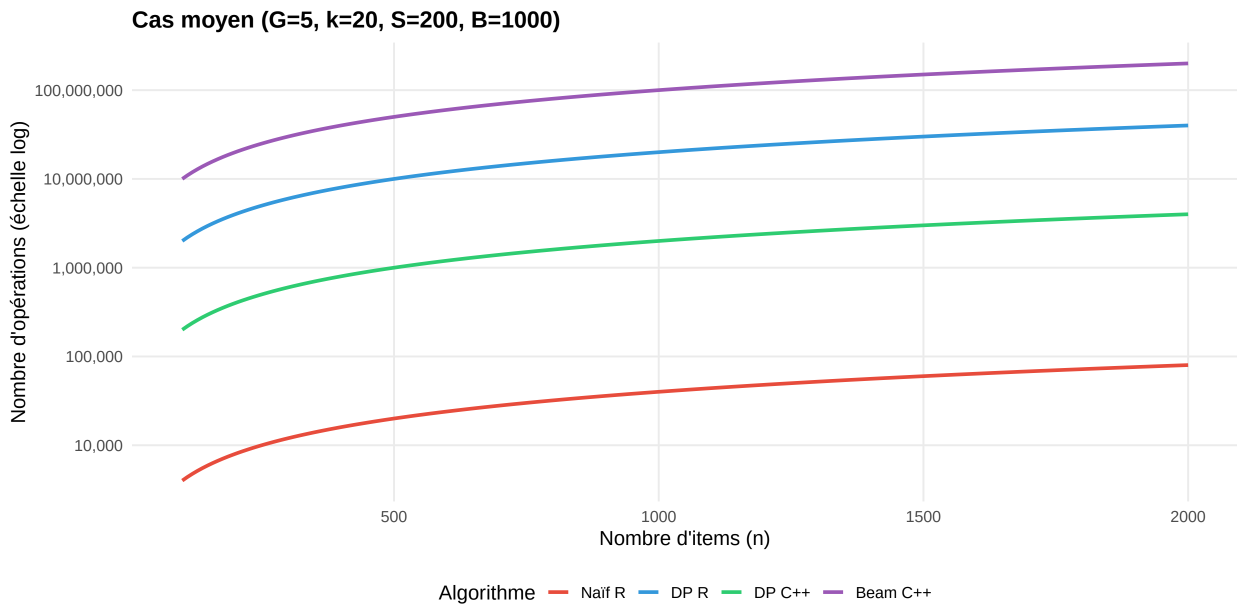
Pire cas : $\boxed{T(n) = O(n \log n + nkBG)}$ (même formule, B peut être grand)

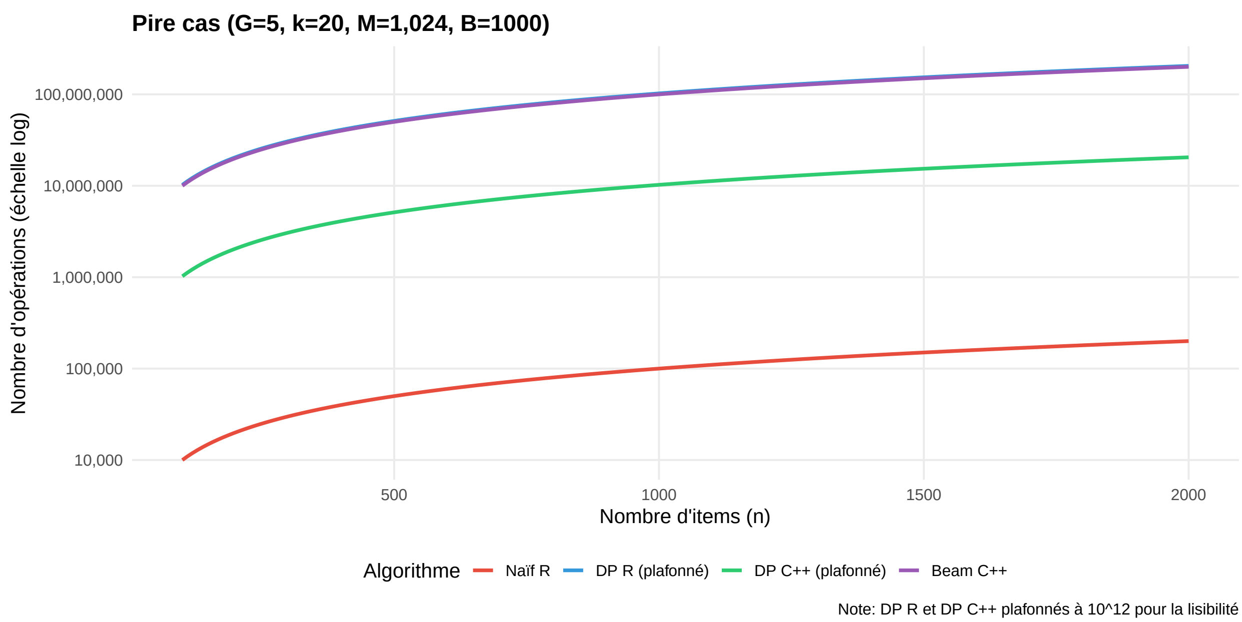
Avantage majeur : Complexité **contrôlable** via `beam_size`, contrairement aux versions exactes.

Tableau récapitulatif

Algorithme	Meilleur cas	Cas moyen	Pire cas
Naïf R	$O(kn)$	$O(kn\bar{g})$	$O(knG)$
DP R	$O(nkG)$	$O(SnkG)$	$O(MnkG)$
DP C++	$O(nkG)$	$O(SnkG)$	$O(MnkG)$
Beam C++	$O(n \log n + nkG)$	$O(n \log n + nkBG)$	$O(n \log n + nkBG)$

Comparaison graphique





Observations

Cas moyen :

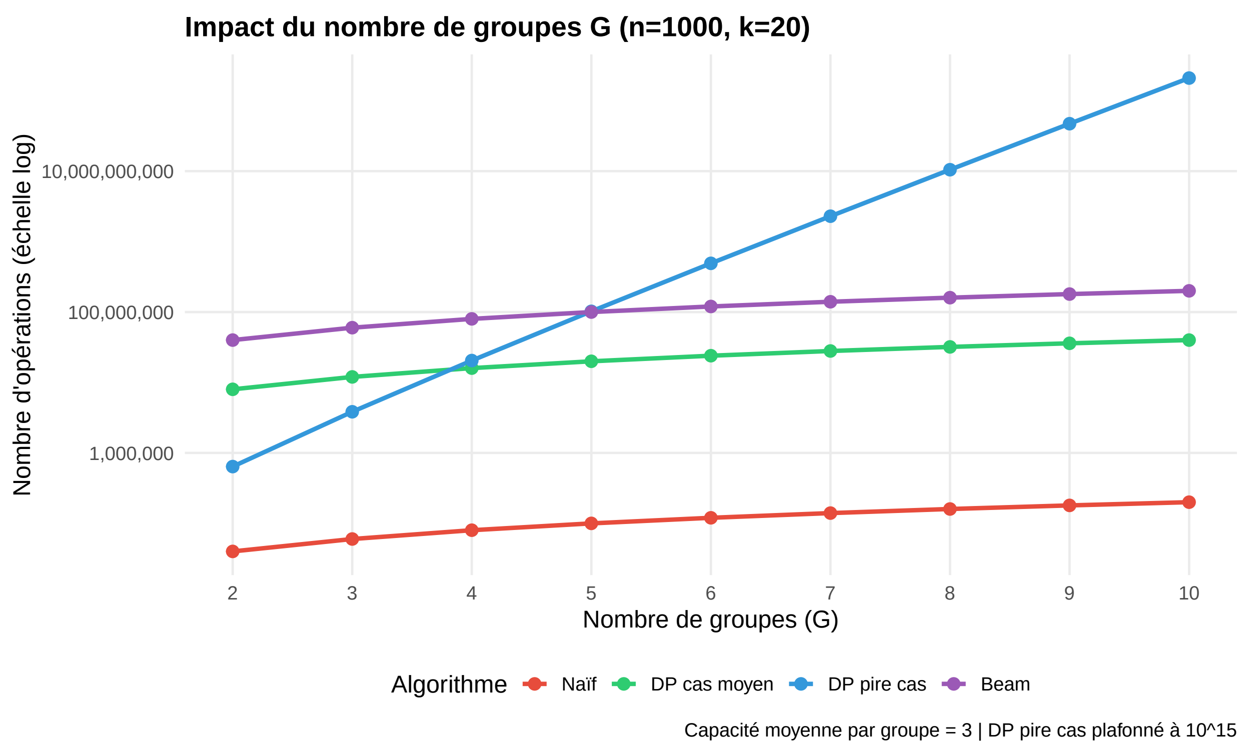
- Beam Search et DP C++ sont très compétitifs
- DP R significativement plus lent (facteur 10)
- Naïf acceptable pour petites instances

Pire cas :

- Explosion exponentielle des DP exact (R et C++)

- Beam Search reste linéaire et prévisible
- Seul algorithme viable pour grands problèmes avec $G \geq 6$

Impact du nombre de groupes G



Analyse

- **G** $\leq$ **4** : Tous les algorithmes sont viables
- **G = 5-6** : DP commence à être problématique en pire cas
- **G >= 7** : Seul Beam Search reste praticable
- L'explosion de M = (cap+1)^G rend DP exact inutilisable pour grand G

Choix de l'algorithme selon le contexte

Contexte	Algorithme recommandé	Justification
Petits problèmes (n<100, G<3, k<10)	**Naïf R** ou **DP R**	Simple, rapide, optimal
Moyens problèmes (n<1000, G<5, k<20)	**DP C++**	Optimal garanti, performance acceptable
Grands problèmes (n>1000 ou G>5)	**Beam C++**	Seul viable, quasi-optimal
Production avec contraintes temps	**Beam C++** (B=5000)	Prédictible, contrôlable
Recherche d'optimum prouvé	**DP C++** si faisable	Exact mais peut échouer

$$\text{Si } M = \prod_{g=1}^{G} (\text{max\_cap}_g + 1) > 100000 \Rightarrow \text{Utiliser Beam Search}$$

7. Temps de calcul