901 — Structures de données : exemples et applications.

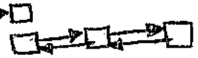
<u>Motivations</u> Confronté à un problème d'algorithmique, les structures de données permettent de manipuler et de stocker les objets que l'on souhaite utiliser.

I - Types de données

1. Définitions

Déf Les types constituent une description du format de représentation interne des données en machine.

<u>Remarque</u>: Les types libèrent le programmeur du souci de la représentation physique des données.

<u>EXEMPLES</u>: Les types de bases donnés par les langages de programmation sont généralement : booléens, entiers, ...
Mais certains langages proposent des types de bases plus avancés comme :
- tableau 
- liste chaînée 
- liste doublement chaînée 

Déf Un type de données abstrait est la description d'un ensemble organisé d'objets et des opérations de manipulation sur cet ensemble.

Description d'un type de données abstrait

signature { - Sorte (noms d'ensembles de valeurs) ; - Opérations (et leurs profils) }
- Préconditions (domaines de définition des opérations)
- Axiomes (propriétés des opérations).

<u>EXEMPLE</u>: DICTIONNAIRE [BBC 52-53] = table

- <u>Sorte</u> dictionnaire ; utilise chaîne de caractères, booléen
- <u>Opérations</u>: • DICOVIDE : → dictionnaire
  - • EST-DICOVIDE : dictionnaire → booléen
  - • CHERCHER ; ch. de car x dictionnaire → booléen
  - • INSERER : ch. de car x dictionnaire → dictionnaire
  - • SUPPRIMER : ch. de car x dictionnaire → dictionnaire

clé ≠ valeur

- <u>Préconditions</u> SUPPRIMER(x, d) est définie ssi CHERCHER(x,d) est vrai. • [...]
- <u>Axiomes</u> • EST-VIDE (DICOVIDE) est vrai [...]

<u>Remarque</u>: Le concept de type abstrait ne dépend pas du langage de programmation.

2. Aspect programmation

Déf Une structure de données est l'implémentation explicite d'un ensemble organisé d'objets, avec la réalisation des opérations d'accès, de construction et de modifications afférentes. [BBC p37]

<u>Remarque</u>: Une structure de données est l'implémentation d'un type de données.

Le concept de type abstrait va permettre d'appliquer une DEMARCHE DESCENDANTE (on conçoit un algorithme à partir de la définition de types de données) par opposition à une DEMARCHE ASCENDANTE (on utilise une représentation concrète des types en termes d'objets du langage utilisé). [FROI p54]

<u>Remarque</u>: La complexité dépend de la structure de données utilisées.

II - Les ensembles [FROI p85...]

Déf Un ensemble est une collection d'objets sans répétition.

<u>Remarque</u>: pas de notion d'ordre sur les objets d'un ensemble.

Description du type

- <u>Sorte</u>: Ensemble ; utilise élément, booléen
- <u>Opérations</u>: • ENSVIDE : → Ensemble
  - • AJOUTER / SUPPRIMER : élément x Ensemble → Ensemble
  - • APPARTIENT : élément x Ensemble → booléen
  - • EST_ ENSVIDE : Ensemble → booléen.
  - • CHOISIR : Ensemble → élément.

Implémentations :

- Tableau de booléens : à chaque élément on fait correspondre une case du tableau.

⊕ efficacité des opérations

⊖ mémoire, ensemble borné de valeurs possibles.

- Liste chaînée :

⊕ mémoire, ensemble infini de valeurs possibles

⊖ opérations plus coûteuses

- Tableau.

Tableau comparatif

	ENSVIDE	EST-ENSVIDE	AJOUT	SUPP.	APPARTIENT	CHOISIR
Tabl. de Booléens	O(N)	O(N)	O(1)	O(1)	O(1)	
Liste chaînée	O(1)	O(1)	O(N)	O(N)	O(N)	

III _ Structures linéaires

1. Listes

Déf **Une liste est une suite finie d'éléments où les insertions et suppressions se font aux extrémités et à l'intérieur de la liste.** [BBC]

Description du type

- Sorte Liste ; utilise élément, booléen, entier
- Opérations
  - LISTEVIDE : → Liste
  - EST_LISTEVIDE : Liste → booléen
  - AJOUT_k_PLACE : élément × Liste × entier → Liste
  - SUPPRIMER_k_PLACE : Liste × entier → Liste
  - ACCEDER_k_ELEMENT : Liste × entier → élément

Implémentation

- Tableau ⊕ accès au k ème élément

⊖ mémoire, ajout et suppression.

- Liste chaînée ⊕ pas de longueur maximum pour la taille de la liste. ⊖ opérations d'accès, ajout et suppression

2. Piles et Files

Déf **Une pile est une suite finie d'éléments avec insertion et suppression du même côté (LIFO)** [BBC]

EXEMPLE : pile de dossiers, d'assiettes

Description du type PILE

- Sorte Pile ; utilise booléen, élément
- Opérations : PILEVIDE, EST-PILEVIDE, SOMMET, EMPILER, DEPILER

Implémentation

- Tableau (avec un indice de sommet de pile)
- Liste chaînée

APPLICATION : Tri par insertion

Déf **Une file est une suite finie d'éléments avec insertion d'un côté, suppression de l'autre (FIFO)**

EXEMPLE file d'attente

Description du type FILE

- Sorte FILE ; utilise booléen, élément
- Opérations FILEVIDE, EST-FILEVIDE, TETE, ENFILER, DÉFILER

Implémentation

- Tableau (avec deux indices) ⊖ taille limitée !
- Liste chaînée (avec deux pointeurs pour désigner la tête et la queue) ou liste circulaire.

IV _ Structures arborescentes

1. Graphe

Déf **Un graphe orienté (resp. non orienté) G est un couple ⟨S, A⟩, S est un ensemble fini de sommets, A un ensemble fini d'arcs (resp. d'arêtes)**

EXEMPLE : réseau routier, ordonnancement de tâches

Description du type

- Sorte Graphe, sommet ; utilise entier, booléen

[FR01] [FR01] [FR01 + 137]

- Opérations GRAPHEVIDE, AJOUTER_SOMMET/ARC, EST_UN_SOMMET/ARC, RETIRER_SOMMET/ARC, d_PREDECESSEUR/SUCCESSEUR, k.ème_PRED/SUCC.

Implémentations

- Matrice d'adjacence $M_{ij} = \begin{cases} 1 \text{ s'il existe un arc entre } i \text{ et } j \\ 0 \text{ sinon} \end{cases}$

⊕ Opérations sur les arcs
⊖ Le nombre de sommets n'évolue pas.

- Liste d'adjacence : à chaque sommet, on associe la liste de ses successeurs rangés dans un certain ordre.

⊕ Mémoire ⊖ Opérations sur les arcs.

Applications :

- parcours de graphe $\begin{cases} \text{en largeur (file)} \\ \text{en profondeur (pile)} \end{cases}$
- calcul de plus court chemin par algorithme de Dijkstra. [DVP 1]

2. Arbres

Def Un arbre est un graphe connexe sans cycle dont l'un des sommets, appelé la racine, est distingué.

EXEMPLE : généalogie, organisation de fichiers systèmes

CAS PARTICULIER Arbres binaires

[FROI 99] Def Un arbre binaire est soit vide, soit de la forme $B = \langle o, B_1, B_2 \rangle$; où B_1, B_2 sont des AB disjoints et 'o' est un noeud appelé racine.

Description du type

- Sorte Arbre, noeud ; utilise élément
- Opérations ARBREVIDE, CONSTRUCTEUR, RACINE, FILS_GAUCHE/DROIT, CONTENU_NOEUD.

Implémentations

- pointeurs : à chaque noeud, on associe deux pointeurs vers les sous arbres gauche et droit
- tableau

APPLICATION Arbres binaires de recherche :

Def Un arbre binaire de recherche est un arbre binaire étiqueté tel que pour tout noeud v de l'arbre : les éléments de tous les noeuds du sous-arbre gauche (resp. droit) de v sont $\leq$ (resp $\geq$) à l'élément contenu dans v [FROI p198]

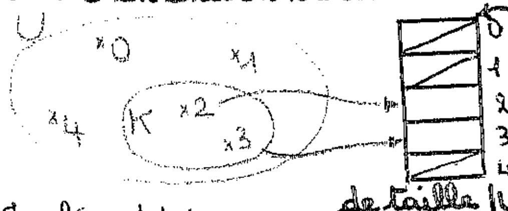
[DVP 2 ABR optimaux]

V - Table de Hachage [FROI]

Une table de hachage est une structure de données permettant d'implémenter efficacement les dictionnaires
les gros ensembles.

1. Tables à adressage direct

Soit U un univers de clé de taille raisonnable. $K \subseteq U$.



Chaque élément correspond à un indice de la table.

Implémentation

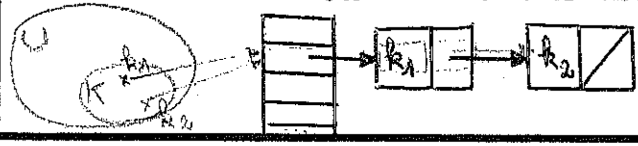
- Tableau de taille $|U|$. ⊕ Chaque opération en $O(1)$

2. Tables de hachage

Si l'univers U est très grand, on définit une fonction de hachage $h : U \to \{0, \dots, m-1\}$ pour une table de hachage $T[0, \dots, m-1]$. h permet de déterminer l'alvéole où sera stockée la clé. Si $m < |U|$, la fonction h n'est pas injective, et lorsque plusieurs clés ont même image par h, on parle de collision.

3. Résolution des collisions par chaînage

Tous les éléments d'une même alvéole sont placés dans une même liste chaînée.



ARBRES BINAIRES DE RECHERCHE OPTIMAUX

PROBLÈME : On cherche à traduire un texte de l'anglais au français.
Pour chaque mot du texte, il faut chercher l'équivalent français.

Idée : Utiliser un arbre binaire de recherche contenant n mots anglais comme clé (les traductions en français sont des données satellites)

- Minimiser le temps total de consultation de l'arbre (car à chaque mot du texte, on va faire une recherche dans l'arbre).
- Gérer le cas des mots anglais dont on ignore la traduction

> Connaissant la fréquence d'apparition de chaque mot, comment organiser un ABR de façon à minimiser le nombre de noeuds visités durant la traduction du texte ? → ABR optimal.

Cadre et notations

→ $K = \{k_1, \dots, k_n\}$ n clés distinctes, triées. (= les mots dont on connaît la traduction)
Chaque clé k_i a une probabilité p_i d'être concernée par la recherche

→ Soient $\{d_0, \dots, d_n\}$ $n+1$ clés factices, représentant les valeurs extérieures à K

- d_0 représente toutes les valeurs $< k_1$
- $\forall i \in \{1, \dots, n-1\}$ d_i représente les valeurs $\in \,]k_i, k_{i+1}[$
- d_n représente toutes les valeurs $> k_n$

Chaque clé d_i a une probabilité q_i d'être concernée par la recherche dans l'arbre.

$$\boxed{\sum_1^n p_i + \sum_0^n q_i = 1}$$

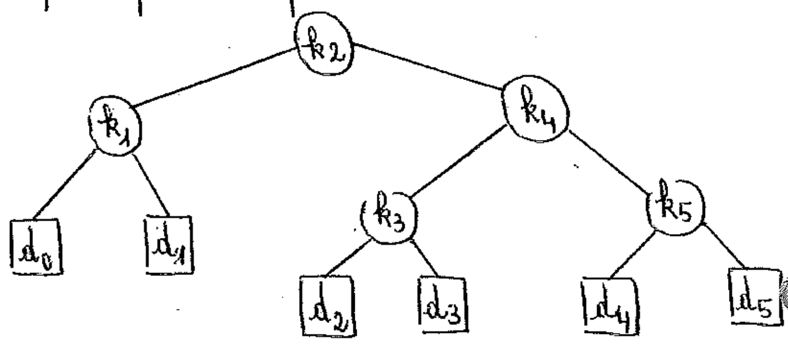
Les k_i sont des noeuds internes, les d_i sont des feuilles.

EXEMPLE pour $n = 5$,

avec les probabilités suivantes

i	0	1	2	3	4	5
p_i		0,15	0,1	0,05	0,1	0,2
q_i	0,05	0,1	0,05	0,05	0,05	0,1

on peut par exemple construire l'arbre suivant



mais est-ce optimal ?

Rappel : Le nombre de noeuds examinés pour chercher le noeud N dans un arbre A est : $1 + \text{profondeur}(N)$.

Définition : Le coût espéré d'une recherche dans un ABR T est défini par :

$$E[T] = \sum_{1}^{n} (1 + \text{prof}_T(k_i))p_i + \sum_{0}^{n} (1 + \text{prof}_T(d_i))q_i$$
$$= 1 + \sum_{1}^{n} \text{prof}_T(k_i)p_i + \sum_{0}^{n} \text{prof}_T(d_i)q_i$$

EXEMPLE pour l'arbre précédent, $E[T] = 2{,}80$

Objectif Pour un ensemble donné de probabilités, on cherche à construire un ABR qui minimise E.

→ La vérification exhaustive de toutes les possibilités ne donne pas un algorithme efficace.

Outil : PROGRAMMATION DYNAMIQUE

• Identification des sous-problèmes.

- Un sous-arbre T' d'un ABRO doit contenir une plage contiguë $k_i, \dots, k_j$ ($1 \leq i \leq j \leq n$) ainsi que les clés factices $d_{i-1}, \dots, d_j$
- T' doit être optimal pour ce sous-problème.

  (Sinon, s'il existait T'' de coût < à celui de T', on pourrait remplacer T' par T'' dans T, ce qui contredirait l'optimalité de T)
- Soit $\{k_i, \dots, k_j\}$ un sous-problème, l'une des clé k_r est la racine du sous-arbre optimal contenant ces clés.

  Le sous-arbre gauche de k_r contient $k_i, \dots, k_{r-1}, d_{i-1}, \dots, d_{r-1}$ (et est réduit à d_{i-1} si $r = i$)

  Le sous-arbre droit de k_r contient $k_{r+1}, \dots, k_j, d_r, \dots, d_j$ (réduit à d_j si $r = j$).

• Formule récursive

Notations : $\begin{cases} e[i,j] : \text{coût espéré de recherche dans un ABRO contenant } k_i, \dots, k_j. \\ w(i,j) = \sum_{i}^{j} p_k + \sum_{i-1}^{j} q_k \end{cases}$

Alors, $e[i, i-1] = q_{i-1}$ (uniquement la clé d_{i-1})

et si k_r est la racine optimale,

$$e[i,j] = p_r + \left(e[i, r-1] + w(i, r-1)\right) + \left(e[r+1, j] + w(r+1, j)\right)$$
$$= e[i, r-1] + e[r+1, j] + w(i,j)$$

Finalement,

$$e[i,j] = \begin{cases} q_{i-1} & \text{si } j = i-1 \\ \min\limits_{i \le r \le j} \{ e[i,r-1] + e[r+1,j] \} + w(i,j) & \text{si } i \le j \end{cases}$$

• **Algorithme**

- Pour stocker les valeurs de $e[i,j]$, on utilise un tableau

$$e[1,\dots,n+1\,;\,0,\dots,n]$$

- Pour éviter de refaire certains calculs et pour les faciliter, on considère également un tableau

$$w[1,\dots,n+1\,;\,0,\dots,n]$$

défini par

$$w(i,j) = \begin{cases} q_{i-1} & \text{pour } j = i-1 \\ w(i,j-1) + p_j + q_j & \text{pour } j \ge i \end{cases}$$

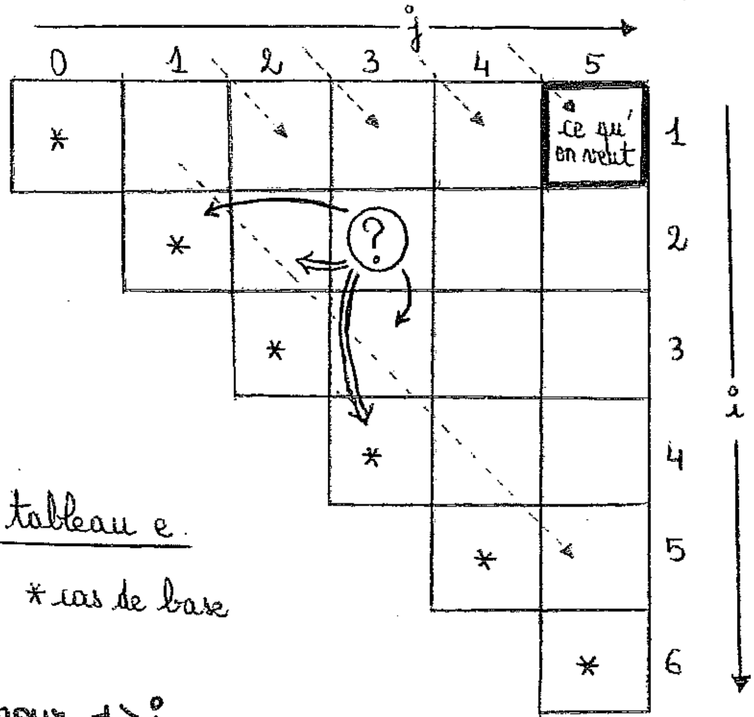


tableau e.

$*$ cas de base

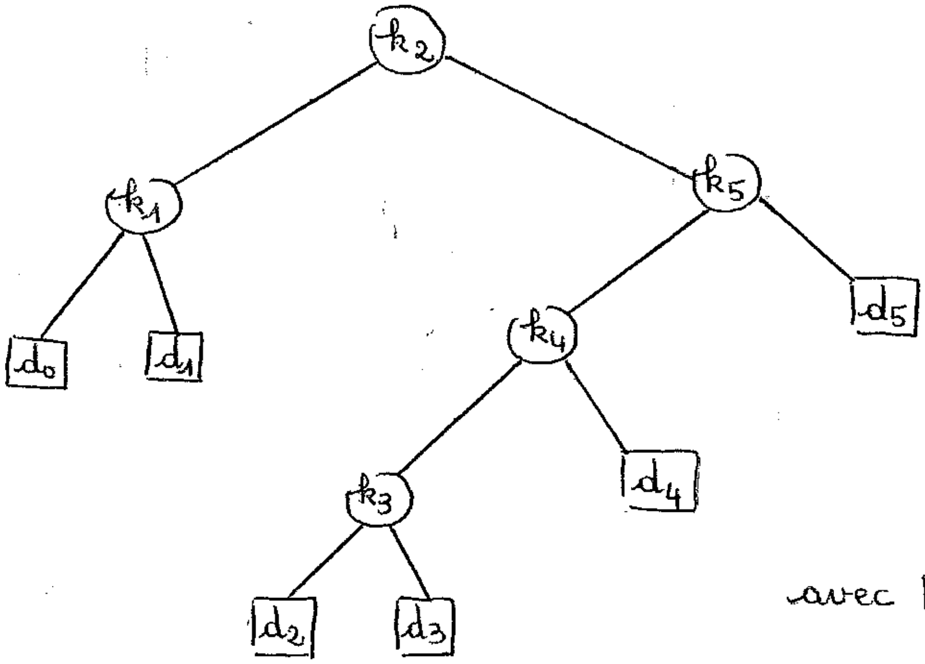
- Enfin, pour gérer la structure, on utilise un tableau racine où racine(i,j) contient l'indice r tel que k_r est la racine de l'ABRO pour le sous-problème $k_i,\dots,k_j$

ABR-OPTIMAL(p,q,n)

```
• créer tableaux e[1,...,n+1 ; 0,...,n] , w[1,...,n+1 ; 0,...,n]
                racine[1,...,n ; 1,...,n]
• pour i = 1 à n+1
      e[i,i-1] = q_{i-1} ;   w[i,i-1] = q_{i-1} ;
• pour l = 1 à n
      pour i = 1 à n-l+1
          j = i+l-1 ;
          e[i,j] = ∞ ;
          w[i,j] = w[i,j-1] + p_j + q_j ;
          pour r = i à j
              t = e[i,r-1] + e[r+1,j] + w[i,j]
              si t < e[i,j]
                  e[i,j] = t ; racine[i,j] = r
• retourner e[1,n] et racine ;;
```

Complexité en $O(n^3)$

EXEMPLE : pour l'échantillon précédent, l'ABRO est



avec $E = 2,75$

Dijkstra

Le but est, étant donné un graphe $G = (S, A)$ orienté dont les arêtes sont étiquetées par des poids entiers ≥ 0 de trouver le plus court chemin d'un sommet s à un sommet v.

→ En fait on calcule le pcc d'un sommet s à tous les autres de G.

Idée : approche gloutonne : on s'intéresse à chaque étape au sommet le plus proche, à l'arc le moins coûteux.

Notation : w fonction de poids des arcs

d : Tableau des distances les plus courtes estimées, ie le plus court chemin possible passant par les sommets déjà traités

π : Tableau des prédécesseurs dans le plus court chemin, ie $\pi[v]$ est le prédécesseur de v dans le chemin de poids $d[v]$ reliant s à v.

- Relâchement :

relâcher (u, v) = tester si on peut améliorer le plus court chemin de s à v en passant par u

Relâcher (u, v, w)
 Si $d[v] > d[u] + w[u, v]$
 alors $d[v] \leftarrow d[u] + w(u, v)$;
 $\pi(v) \leftarrow u$;

- Extraire-Min (F, d) renvoie le sommet de F de distance minimale et le supprime de F

(F : ensemble de sommets de G)

Algo

Dijkstra (G, w, s)

E ← ∅ "E utile pour la preuve de l'algo"

F ← S

$O(|S|)$

Pour tout $v \in S \setminus \{s\}$

faire $d[v] \leftarrow \infty$

$\pi[v] \leftarrow$ ~~~~ v

$d[s] \leftarrow 0$

Tant que $F \neq \emptyset$

on choisit le sommet le plus proche de ceux déjà traités ⇒ approche gloutonne

faire $v \leftarrow$ Extraire-Min (F, d)

$E \leftarrow \{v\} \cup E$

pour chaque successeur de v

on met à jour la distance des successeurs de v

faire Relâcher (u, v, w)

Renvoyer π et d

Terminaison : A chaque étape on retire qqun de F de taille finie

Correction :

1er invariant de boucle : $F = S \setminus E$ OK

car le sommet supprimé de F est ajouté à E.

2ème invariant de boucle :

Avant chaque itération du tant que, on a :

$\forall v \in E, \; d[v] = \delta(s, v) \longrightarrow$ distance minimale de s à v

preuve : cela revient à montrer que $d[v] = \delta(s, v)$ lorsque v est ajouté à E (car une fois qu'on a δ on ne change plus car $d[v] \geq \delta(s, v)$ tt le tps)

Par l'absurde : supposons que $\delta(s, v) \neq d[v]$ lorsque v est ajouté à E. Prenons v le premier sommet tel que $\delta \neq d$ à l'ajout

$\Rightarrow$ Avant l'ajout de u à E

- $u \neq s$ car $d[s] = 0 = \delta(s,s)$ dès le début
  $\Rightarrow$ $E \neq \emptyset$ car s est le 1er sommet que l'on ajoute
- si u n'est pas accessible depuis s
  $\Rightarrow$ $\delta(s,u) = d[u] = \infty$ tel que ...
  contradiction
- si il existe un chemin p de s à u
  Soit p un plus court chemin de s à u
  avant ajout de u p relie un sommet de E à un sommet de $S \setminus E$
  $\Rightarrow$ $\exists y \in S \setminus E$ premier sommet tel que $y \in S \setminus E$
  soit x le prédécesseur de y dans p

on a $s \overset{p_1}{\leadsto} x \longrightarrow y \overset{p_2}{\leadsto} u$

Mq $d[y] = \delta(s,y)$ lorsque u est ajouté à E

$x \in E \Rightarrow x \neq u$ et donc $\delta(s,x) = d[x]$
car u est le premier elt de E qui ne vérifie pas cela

quand x a été ajouté, l'arc (x,y) a été relâché
et comme p est plus court chemin de s à u
$p_1 : s \leadsto x \to y$ est plus court chemin de s à y
et donc le relâchement a mis à jour $d[y]$ en
$d[x] + w(x,y)$
$= \delta(s,x) + w(x,y)$
$= \delta(s,y)$

Donc $\Rightarrow$ $d[y] = \delta(s,y)$ à l'ajout de u

$\Rightarrow$ $d[y] = \delta(s,y) \leq \delta(s,u) < d[u]$ par hyp $\Rightarrow d[y] < d[u]$
(car poids ≥ 0)

Or lors de l'ajout, $y \notin E$
$\Rightarrow$ $d[u] \leq d[y]$ contradiction
par déf de l'ajout

~~Donc plus~~

Les 2 invariants concluent car à la fin $E = S$

Chaque arc est parcouru une unique fois

<u>Complexité</u> :

- F implementé par un tableau / Liste / Pile
  ↳ Extraire-Min ⇝ $O(|S|)$ et appelé $|S|$ fois
  $\Rightarrow O(|S|^2)$
  ↳ Relâcher ⇝ $O(1)$ appelé $|A|$ fois
  $\Rightarrow O(|A|)$

$\Rightarrow O(|S|^2 + |A|) = O(|S|^2)$

- F par un tas
  ↳ Extraire-Min ⇝ $O(\log |S|)$ appelé $|S|$ fois
  ↳ Relâcher ⇝ $O(\log |S|)$ appelé $|A|$ fois

$\Rightarrow O((|S| + |A|) \log |S|) \;(= O(|A| \log S))$
↳ mieux si graphe peu dense