Leçon 921 : Algorithmes de recherche et structures de données associées.

Julie Parreaux

2018 - 2019

Références pour la leçon

[1] Beauquier, Berstel et Chretienne, *Éléments d'algorithmique.*
[2] Cormen, *Algorithmique.*
[4] Froidevaux, Gaudel et Soria, *Types de données et algorithmes.*
[5] Knuth, *The Art of Computer Programming, vol 3*

Développements de la leçon

Les B-arbres

L'automate minimale pour la recherche de motifs

Plan de la leçon

Motivation

Défense

Ce qu'en dit le jury

Le sujet de la leçon concerne essentiellement les algorithmes de recherche pour trouver un élément dans un ensemble : l'intérêt des structures de données proposées et de leur utilisation doit être argumenté dans ce contexte.

La recherche d'une clé dans un dictionnaire sera ainsi par exemple l'occasion de définir la structure de données abstraite « dictionnaire », et d'en proposer plusieurs implantations concrètes. De la même façon, on peut évoquer la recherche d'un mot dans un lexique : les arbres préfixes (ou digital tries) peuvent alors être présentés. Mais on peut aussi s'intéresser à des domaines plus variés, comme la recherche d'un point dans un nuage (et les quad-trees), et bien d'autres encore.

Introduction

Motivation : Stocker et exploiter des données est une utilisation des plus commune de l'informatique. Savoir retrouver une donnée précise dans une collection est un problème récurent. Cas de la suppression

Définition du problème :

entrée K une collection, $E \subseteq K$ une sous-collection finie, on note $n = |E|$, et $k \in K$

sortie $k \in E$?

L'élément que l'on cherche doit être bien "typé".

Recherche associative (ne porte que sur la valeur de la clé) ou non

1 Recherche dans un dictionnaire

Objectif : Recherche d'un élément quand $K = \{(cle, valeur)\}$ et E est un dictionnaire (au sens de la structure abstraite).

Principe de la solution : On se ramène à un problème de recherche dans l'ensemble des clés. Les structures de données utilisées pour stocker les clés vont donc être à la base de l'implémentation de notre dictionnaire. On implémente le dictionnaire avec ses données satellitaire : stocké à part des clés et accessible par un pointeur.

Dans cette leçon, nous allons pas traiter le cas du nombre d'occurrence. Les algorithmes que nous présentons peuvent être "facilement" adapté pour ce problème (il faut les faire continuer à la place de s'arrêter comme nous le faisons).

1.1 La recherche suit une loi uniforme sur l'ensemble des clés

Objectif : Recherche d'un élément quand $K = \{(cle, valeur)\}$ où les clés sont à valeur dans un ensemble et E est un dictionnaire (au sens de la structure abstraite) et k est uniformément distribué sur E (ou qu'on ne sait pas s'il existe une distribution non-uniforme).

On veut une complexité de la recherche qui ne dépend pas de la position de la clé. Les structures de données associées doivent être équilibré : la place de la donnée recherché ne doit pas influencer sur le temps de recherche.

Recherche naïve : On compare k à tous les éléments de E .

— Structure de données associée : liste tableau

— Complexité temporelle dans le pire cas : $O(n)$

— + : facilité de la maintenance de la structure

— - : complexité temporelle de la recherche

Recherche dichotomique [4, p.178] :

Algorithm 1 Recherche dichotomique dans un tableau trié

```
1: function DICHO(k, E, g, d, res)  ▷ Entré : g, d, res ∈ {0, 1, ..., n} ; Sortie : 0 si l'élément n'est pas dans le tableau ; {1, ..., n} l'indice de l'élément k dans le tableau sinon.
2:     if g ≤ d then
3:         m ← (g + d)div 2
4:         if k = E[m] then
5:             res ← m
6:         else
7:             if k < E[m] then
8:                 DICHO(k, E, g, m − 1, res)
9:             else
10:                DICHO(k, E, m + 1, d, res)
11:            end if
12:        end if
13:    end if
14:    res ← 0
15: end function
```

— Structure de données associée : utilisation d'un tableau **trié**

— Complexité temporelle dans le pire cas : $O(\log n)$

— + : complexité de la recherche

— - : on doit triée et maintenir **triée** le tableau

Pour aller plus loin : la recherche par interpolation [4, p.190]. C'est dichotomie amélioré dans laquelle on ne coupe pas l'ensemble des données en deux ensembles de tailles égales, mais on le divise suivant la place approximative de la données que l'on cherche. La recherche dans un dictionnaire (de la langue française) : on l'ouvre plutôt au début quand on cherche le mot *ananas*.

La recherche dichotomique se rapproche à une recherche dans un ABR. Les ABR sont des structures de données facilitant l'implémentation du principe de la dichotomie (notamment dans le maintient de la structure).

Recherche supportée par un ABR équilibré : recherche dans un ABR.

Définition [4, p.197] : Un arbre binaire de recherche (ABR) est un arbre binaire étiqueté tel que pour tout nœud x d'étiquette e de l'arbre :

— les étiquettes de tous les nœuds du fils gauche de x sont inférieur à e ;

— les étiquettes de tous les nœuds du fils droit de x sont supérieurs à e ;

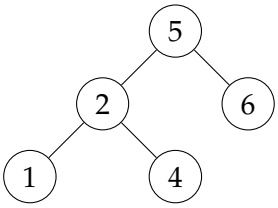


FIGURE 1 – Un exemple d'ABR équilibré

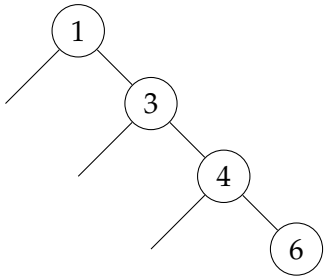


FIGURE 2 – Un exemple d'ABR déséquilibré

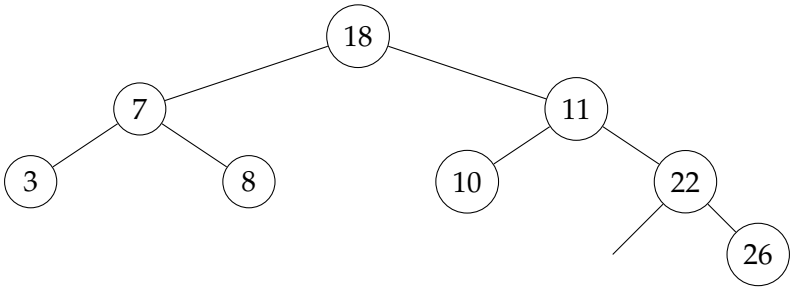


FIGURE 3 – Un exemple d'arbre AVL

exemple figures 1 et 2.

Définition [4, p.221] :Un ABR est dit équilibré si sa hauteur est une fonction logarithmique en sa taille.

Implémentations : AVL [4, p.224] (Figure 3), arbre rouge-noir [2, p.287] (Figure 4) DEVs

Complexité temporelles dans le pire cas : $O(\log n)$

\+ : complexité de la recherche

Quelques exemples en géométrie algorithmique :

— *Problème* : L'intersection de s droites par la méthode de balayage. On cherche un élément où K l'ensemble des points du plan et E l'ensemble des points d'intersection de deux droites.

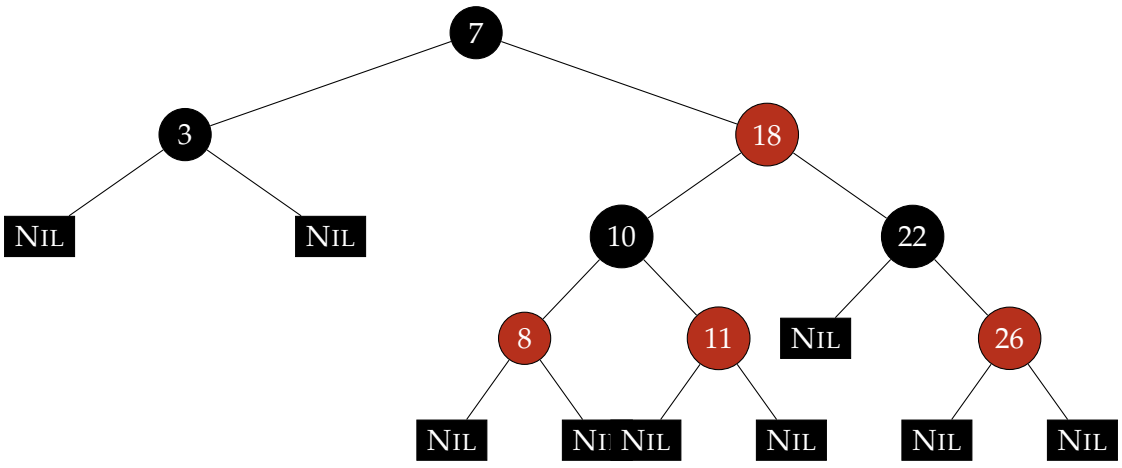


FIGURE 4 – Un exemple d'arbre rouge-noir

Structure	Insertion	Suppression (sans recherche)	Recherche
Tableau (non triée)	$O(n)$	$O(n)$	$O(n)$
Tableau (triée)	$O(n \log n)$	$O(n)$	$O(\log n)$
Liste	$O(1)$	$O(1)$	$O(n)$
Liste triée	$O(\log n)$	$O(1)$	$O(\log n)$
ABR	$O(n)$	(dépend de la maintenance)	$O(n)$

TABLE 1 – Complexité dans le pire cas de la manipulation de ces structures de données.

Complexité pour trouver une intersection : $O(s \log s)$ dans le pire cas.
Implémentation : ABR équilibré (arbre rouge-noir)
Dans ce cas la recherche permet de mieux décrire l'ensemble E.

— *Problème* : La recherche d'un point dans un nuage. On cherche un élément où K est l'ensemble des points du plan et E est l'ensemble des points du nuage.
Structure de données associées : quad-tree pour réaliser un maillage uniforme du rectangle enveloppant le nuage de points.
Pour aller plus loin : On peut également utiliser des *kd*-arbres pour faire le maillage du plan.

Les complexités de maintenance des structures de données étudiés sont données dans la Table 1. On a un compromis entre la complexité pour ajouter et celle pour la recherche. On choisit la structure de donnée en fonction de la propriété dynamique des données.

1.2 La recherche n'est pas uniforme sur l'ensemble des clés

Constations : Il existe des clés qui appairassent plus souvent que d'autre dans les requêtes de la recherche : pour celles-ci nous devons minimiser les temps de recherche.
Objectif : Recherche d'un élément quand $K = \{(cle, valeur)\}$ où les clés sont à valeur dans un ensemble et E est un dictionnaire (au sens de la structure abstraite) et k suit une distribution non-uniforme sur E.
Les complexité pour atteindre les clés les plus fréquentes doit être minimale.

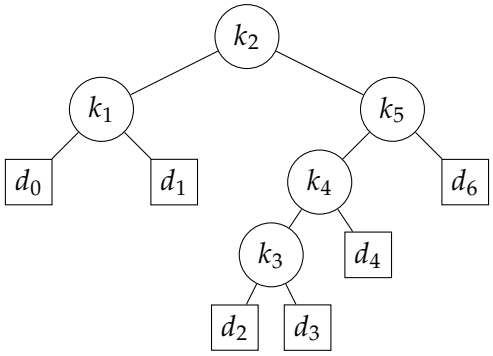
Recherche auto-adaptative : mettre en première position l'élément que l'on vient de rechercher.

— Structure de données associée [4, p.175] : liste
— Remarque culturelle : complexité temporelles dans le pire cas est en $O(\frac{n}{\log n})$ [5] (quand il y a beaucoup de recherches)

\+ : on n'a pas besoin de connaître la distribution des clés
Il existe une variante pour les tableaux : elle consiste à échanger l'élément que l'on vient de rechercher avec l'élément qui le précède (si cela est possible).

Recherche supportée par un ABR optimal [2, p.368] : recherche dans un ABR DEV

— Pour construire l'arbre, on a besoin de connaître la distribution de probabilité. De plus, cet arbre n'est pas dynamique : on n'ajoute ni ne supprime aucune clé sans recalculer entièrement l'arbre
— Implémentation : arbre splay



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	0.1	0.2

FIGURE 5 – Exemple d'un ABR optimal qui n'est pas équilibré

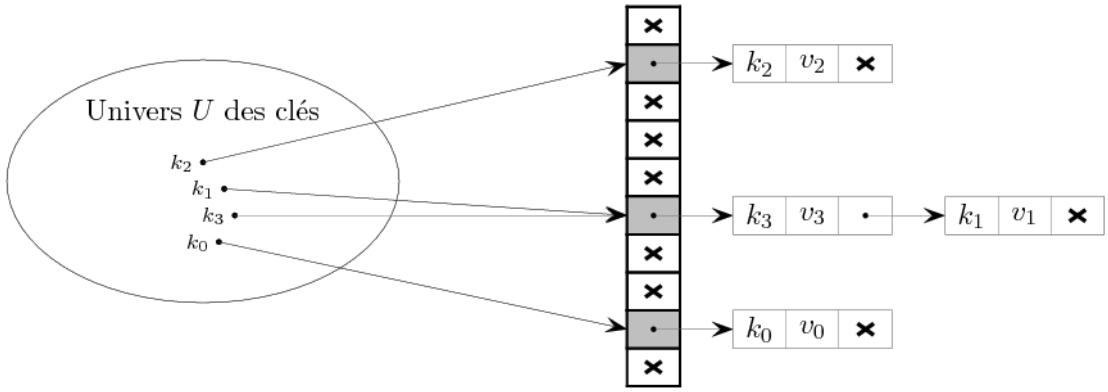


FIGURE 6 – Exemple d'une table de hachage dont la gestion des collisions se fait par chaînage simple.

Définition : Un ABR optimal minimise la complexité espérée de la recherche en fonction de la distribution sur les clés (Figure 5).

Limites pour les deux sous-parties précédentes : Il nous faut un ordre sur les clés. De plus, ces structure l'accès concurrent n'est pas efficace. En effet, lors d'accès concurrents, le sous-arbre enraciné en le noeud sur lequel on ait ne peut plus être partager (des fois qu'on modifierait le sous-arbre).

1.3 La recherche s'effectue sur un ensemble sur lequel on ne souhaite pas utiliser son ordre total

Objectif : Recherche d'un élément quand $K = \{(cle, valeur)\}$ pour lequel on ne veut pas travailler avec l'ordre totale des clés. les images, les graphes, ...

La recherche ne doit pas perturbée par l'ordre (ou le manque d'ordre) sur les clés.

Principe : Pour chaque valeur de clé, une fonction de hachage h lui associe un entier représentant un indice d'un tableau qui va le stocker.

Définition : Deux éléments distincts u et v sont en collisions si et seulement si $h(u) = h(v)$.

Implémentation : table de hachage [2, p.238]

— Doit gérer des collisions : chaînage simple (Figure [1] 6)

— Complexité de la recherche en $O(n)$ dans le pire cas.

1. Source de l'image http ://gallium.inria.fr/ maranget/X/421/poly/assoc.html

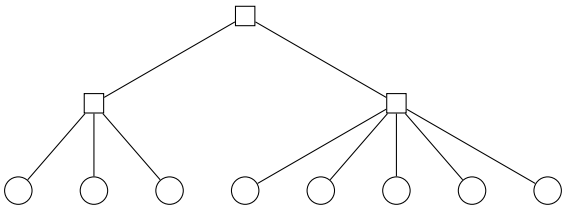


FIGURE 7 – Un exemple de B-arbre de borne minimale $t = 3$.

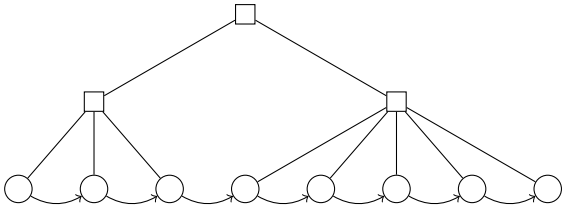


FIGURE 8 – Un exemple de B^+-arbre de borne minimale $t = 3$.

Remarque : Le choix de la fonction de hachage influe sur la complexité de la recherche qui dépend du nombre de collision.

Définition [2, p.258] : Une fonction de hachage est dite parfaite si la recherche à l'aide de cette fonction s'effectue en $O(1)$. DEV

Plus : La table de hachage peut être utilisée de manière concurrente : on ne bloque que la case du tableau que l'on vient d'atteindre donc si la fonction n'est pas trop mauvaise, la majorité de nos clés restent accessibles.

1.4 Le dictionnaire doit être stocké sur un disque externe

Objectif : Recherche d'un élément quand $K = \{(cle, valeur)\}$ où les clés sont à valeur dans un ensemble et E est un dictionnaire qui ne peut pas être dans la mémoire vive.

Problème : Dans ce contexte, le travail en mémoire vive est négligeable devant les accès au disque externe (en ms). Il nous faut donc minimiser le nombre d'accès au disque externe afin d'améliorer la complexité de la recherche. Exemple d'un cas de complexité non temporelle (on regarde le nombre d'accès au disque).

Définition [2, p.447] : Un B-arbre (Figure 7) est un arbre de recherche T qui vérifie les propriétés suivantes :

- Pour chacun des nœuds x sont tels que :
  - $x.e$ est le nombre de clés dans le noeud ;
  - $x.cle_i < \ldots < x/cle_e$ l'ensemble de ces clés ;
  - $x.feuille$ qui vaut vrai si le noeud est une feuille ;
  - $x.c_1, \ldots x.c_{e+1}$ ses fils (si $\neg x.feuille$) et tels que $k_1 < x.c_1 < \ldots < k_e < x.c_e < k_{e+1}$ où k_i est une étiquette du sous-B-arbre enraciner en $x.c_i$;
- L'ensemble des feuilles sont à une même profondeur h ;
- $t \geq 2$ est la borne minimale de T : pour tout noeud x de T différent de la racine, on a : $t - 1 < x.e < 2t - 1$; pour x la racine de T, $1 < x.e < 2t - 1$.

Proposition La recherche, l'insertion et la suppression d'un B-arbre se fait en $O(log_t n)$ d'accès au disque externe. DEV (Nous allons étudier la recherche et l'insertion de cette structure.)

Remarque : Un B-arbre peut être étendu en un B^+-arbre (Figure 8) pour facilité les recherches séquentielles.

Utilisation : Dans les bases de données relationnelles.

2 Recherche d'une chaîne de caractères (motif) dans un texte

Objectif : Recherche d'un élément lorsque K est l'ensemble des mots de $|\Sigma|^*$ (où Σ est un alphabet) et E est l'ensemble des facteurs d'un texte.

Un problème analogue : le nombre d'occurrence.

Applications : Recherche d'un mot dans un ficher texte, d'une chaîne de caractères dans séquence d'ADN, ...

Structure de données associée : tableau de longueur la taille de la chaîne de caractère considérée. On notera $T = [1 \dots t]$ le tableau du texte et $P = [1 \dots p]$ le motif rechercher dans T où t est la taille du texte et p est celle du motif.

Remarques préliminaires : Le motif ne peut être contenue dans le texte uniquement si sa taille est inférieure à celle du texte et s'ils sont définis sur un alphabet commun que l'on note Σ. Ce que l'on supposera dans la suite.

Algorithme naïf [2, p.905] :

Algorithm 2 Recherche naïve d'un motif dans un texte.

```
1:  function RECHERCHE-NAIVE(T, P)   ▷ Entré : T le texte et P le motif ; Sortie : le décalage
    représentant le début du motif dans le texte ou 0 si le motif n'apparaît pas dans le texte.
2:      t ← T.longueur
3:      p ← P.longueur
4:      for s = 0 à t − p do
5:          if P[1, ..., m] = T[s + 1, ..., s + m] then
6:              Retourner s
7:          end if
8:      end for
9:      Retourner 0
10: end function
```

Complexité dans le pire cas : $O((t-p+1)p)$ ou l'ensemble des positions possibles pour le motif et la taille du motif que nous devons tester à chaque fois. Borne atteinte avec $T = a^{p-1}ba^p$ et $P = a^p$.

2.1 Algorithme de Rabin-Karp [2, p.905]

Hypothèse : Le motif est fixe et connu à l'avance.

Hypothèse : On considère que chacun des caractères est exprimé comme un chiffre en base $d = |\Sigma|$. Dans le cadre de cette leçon, on considère que $\Sigma = \{0, 1, \dots, 9\}$

Les valeurs pour le texte sont calculées à la volée lors de la recherche. Le prétraitement permet de calculer la valeur du motif et la première valeur pour le texte.

Complexité du prétraitement : $\Theta(p)$

Complexité dans le pire cas : $O((t-p+1)p)$ Borne atteinte avec $T = a^{p-2}ba^{p-1}$ et $P = ba^{p-1}$. Comme dans le cas naïf.

Application : Le problème de recherche multiples motifs se résout avec un lourd pré-traitement sur le texte T ou les multiples motifs P_i afin de comparer les multiples P_i à une portion de texte une seule fois. On adapte l'algorithme de Rabin-Karp pour résoudre efficacement ce problème. (Autre algorithme pour ce problème : l'algorithme d'Aho-Corasick.)

Utilisation : détection de plagiat ; comparaison d'un fichier suspect à des fragments de virus ; ...

2.2 Algorithme de Knuth-Morris-Pratt [2, p.905]

Hypothèse : Le motif est fixe et connu à l'avance.

Définition : La fonction suffixe associé au motif P $\sigma : \Sigma^* \mapsto \{0, 1, \dots, m\}$ donne la longueur du plus long préfixe de P qui est suffixe de $x \in \Sigma^*$.

Algorithm 3 Algorithme de Rabin-Karp pour la recherche d'un motif dans un texte.

function RABIN-KARP(T, P, d, q) ▷ Entré : T le texte, P le motif, $d = |\Sigma|$, q un nombre premier ; Sortie : le décalage représentant le début du motif dans le texte ou 0 si le motif n'apparaît pas dans le texte.
    $t \leftarrow T.longueur$; $p \leftarrow P.longueur$
    $h \leftarrow d^{p+1} mod\ q$
    $p \leftarrow 0$; $t_0 \leftarrow 0$
    for $i = 1$ à p **do** ▷ Pré-traitement
        $p \leftarrow (dp + P[i]) mod\ q$ ▷ Calcul de la valeur du motif
        $t_0 \leftarrow (dt_0 + T[i]) mod\ q$ ▷ Calcul de la valeur du correspondant au 1er motif
    end for
    for $s = 0$ à $t - p$ **do** ▷ Recherche de la correspondance
        if $p = t_s$ **then**
            if $P[1, \ldots, p] = T[s+1, \ldots, s+p]$ **then**
                Retourner s
            end if
            $t_{s+1} \leftarrow (d(t_s - T[s+1]h) + T[s+m+1]) mod\ q$
        end if
    end for
    Retourner 0
end function

Définition : On définit l'automate des occurrences associé à $P[1 \ldots m]$ comme suit : $\mathcal{Q} = \{1, \ldots m\}$; $q_0 = 0$; $F = \{m\}$; $\delta(q, a) = \sigma(P_q a)$ pour tout $a \in \Sigma$ et $q \in \mathcal{Q}$.

Algorithm 4 Calcul de la fonction de transition δ de l'automate des occurrences.

function CALCUL-$\delta(P, \Sigma)$ ▷ Entré : P le motif et Σ l'alphabet que nous utilisons ; Sortie : la fonction transition δ de l'automate des occurrences.
    $p \leftarrow P.longueur$
    for $q = 0$ à p **do**
        for $a \in \Sigma$ **do**
            $k \leftarrow \min(p+1, q+2)$
            repeat
                $k \leftarrow k + 1$
            until P_k préfixe de $P_q a$ ▷ P_q est le préfixe de P de longueur q.
            $\delta(q, a) \leftarrow k$
        end for
    end for
    Retourner δ
end function

Complexité temporelle du prétraitement dans le pire cas : $O(p|\Sigma|)$ Se fait une fois par motif.
Complexité temporelle dans le pire cas : $\Theta(t)$
Définition : La fonction préfixe associé au motif P $\pi : \{1, 2, \ldots, p\} \mapsto \{0, 1, \ldots, p-1\}$ donne la longueur du plus long préfixe de P qui est suffixe propre de P_q où $q \in \{1, 2, \ldots, p\}$.
Si on ne fait pas l'hypothèse d'un préfixe propre alors le plus long suffixe d'un mot qui est aussi son préfixe est le mot en question. De plus, sans cette hypothèse l'algorithme de Knuth-Morris-Pratt serait soit incorrect soit non terminal.

Algorithm 5 Calcul de la recherche dans l'automate des occurrences.

```
1: function RECHERCHE-AUTOMATE(T, δ, p)        ▷ Entré : T le le texte, δ la fonction transition calculée précédemment et p la longueur du motif ; Sortie : le décalage représentant le début du motif dans le texte ou 0 si le motif n'apparaît pas dans le texte.
2:     t ← T.longueur
3:     for i = 1 à t do
4:         q ← δ(q, T[i])
5:         if q = p then
6:             Retourner i − m
7:         end if
8:     end for
9:     Retourner 0
10: end function
```

Algorithme	Prétraitement	Recherche
Naïf	0	$O((t-p+1)p)$
Rabin-Karp	$\Theta(p)$	$O((t-p+1)p)$
Automate fini	$O(p\lvert\Sigma\rvert)$	$\Theta(t)$
Knuth-Moris-Pratt	$\Theta(p)$	$\Theta(t)$
Boyer-Moore	0	$O(tp)$

TABLE 2 – Récapitulatif des complexités pour les algorithmes de recherche de motifs.

Algorithme de Knuth-Morris-Pratt DEV

Idée : Construction intelligente de l'automate précédent pour que la fonction δ puisse être calculer à la volée.

Complexité du prétraitement : $\Theta(p)$

Complexité : $\Theta(t)$

2.3 Pour aller plus loin : algorithme de Boyer et Moore [1, p.358]

Principe : on fait la vérification du motif de la droite vers la gauche. En cas d'erreur et si le caractère d'erreur n'est pas dans le motif, on décale le motif de p caractères dans le texte. En effet, si une erreur de ce type apparaît lors de la vérification, on sait que le motif n'est pas dans le texte vérifié.

Exemples

Cas 1	Cas 2	Cas 3	Cas 4
stupid_spring_string	stupid_spring_string	stupid_spring_string	stupid_spring_string
string	. . . . . string	. . . . . string	. . . . . string

Dans le cas 1 et 3 : le caractère d'erreur ne se trouve pas dans le motif. On peut sauter directement dans le texte les 6 caractères du motif et recommencer par la fin du motif.

Dans le cas 2 et 4 : le g de la clé ne correspond pas au n du texte. Cependant, le motif contient un n, un caractère avant, on peut donc décaler d'une position, et vérifier à nouveau en commençant par la fin du motif.

Complexité : $O(pt)$

Ouverture

— Recherche du min et max dans les différentes structures
Problème : K est un ensemble ordonné et $E \subseteq K$. On cherche $k = \min_K E$ ou $k = \max_K E$.

— Union-find et partition d'un ensemble
Problème : K est l'ensemble des entiers, $E = \{(n, \bar{n})\}$ est un ensemble d'entier avec son représentant de classe d'équivalence.

— Problème de la recherche non-associative : on fait une recherche pas nécessairement sur la valeur clé mais également sur son contexte : recherche "temporelle" (l'élément qui est dans la structure depuis le plus longtemps), du kième élément, selon un autre attribut (mettant un contexte) ...
Application : Base de données.

— Problème d'accessibilité dans un graphe
Problème : K est l'ensemble des états d'un graphe et E est l'ensemble des états accessibles de ce graphe.
Dans ce cas la recherche permet de mieux décrire l'ensemble E.

— *Pour aller plus loin* : Page rank

— *Pour aller plus loin* : Algorithme du simplexe

Recherche géométrique dans le plan

Objectif : Recherche d'un élément lorsque E est un ensemble de point du plan.
Remarque : En dimension supérieure les algorithmes fonctionnent de la même façon mais pour simplifier l'écriture nous travaillons dans le plan.

Recherche de points dans un nuage

Objectif [6, p.36] : Soit $E = (x_i, y_i)_{i \in I}$ un ensemble de point du plan relativement proche. On veut connaître $|I|$.
Solution [6, p.36] : Faire un maillage d'un rectangle (axe parallèle aux cordonnées) contenant l'ensemble des points de I tel que le nombre de maille soit égale au cardinal de I.
Structure de donnée associée [3, p.309] : l'arbre quaternaire
Définition [3, p.309] : Un arbre quaternaire permet de faire une subdivision récursive de l'espace à l'aide de rectangles. Chacun des nœuds de l'arbre possède quatre fils décrivant les quatre subdivisions (uniforme) et les coordonnées d'un point. dessin
On peut mettre un attribue supplémentaire dans les feuilles marquant la présence ou non d'un point dans le noeud.
Principe de l'algorithme : On construit l'arbre quaternaire de telle sorte que chacun des nœuds contient au plus un point de E puis on parcours l'arbre pour compter le nombre de points.
Complexité [3, p.311-321] : $O((dn)^2)$ où $n = |I|$ et d est la hauteur de l'arbre car on construit l'arbre en $O(dn)$ et on effectue un parcourt de l'arbre en $O(dn)$ car il possède $O(dn)$ nœuds.

Recherche des plus proches voisins

Objectif : Soit $E = (x_i, y_i)_{i \in I}$ un ensemble de point du plan relativement proche. On veut trouver les plus proche voisins d'un points donné.

Simplification du problème : On se donne un point et une direction (NordEst, NordOuest, SudEst, SudOuest) et on veut trouver son plus proche voisin.

Solution : On construit l'arbre quaternaire et on fait un recherche dans l'arbre suivant la direction jusqu'à une feuille : si la feuille contient un point on le retourne.

Complexité : $O(dn)$ pour construire l'arbre et en $O(d)$ pour la recherche.

Définition [6, p.243] : Un diagramme de Voronoi diagramme généralisé d'ordre k sur un ensemble fini de point S dans le plan est définit tel que $Vor_k(S) = \cup V(T)$ tel que $T \subseteq S, |T| = k$ et $V(t) = \{p : \forall v \in T, \forall w \in S - T, d(p,v) < d(p,w)\}$.

Moralement il permet de construire des partitions de k plus proche voisin

Principe : on construit un diagramme de Voronoi d'ordre k puis on cherche le point dans ce diagramme.

Complexité du pré-traitement[6, p.252] : $O(k^2 N \log N)$

Complexité[6, p.252] : $O(\log N + k)$

Recherche de deux points les plus proches pour la distance euclidienne [2, p.956]

Objectif : Recherche d'un élément lorsque $E = (x_i, y_i)_{i \in \{1,\dots,n\}}$ avec $n \geq 2$ est un ensemble de point du plan "assez" proche.

Algorithme naïf : tester toute les paires de points

Complexité : $\Theta(n^2)$

Algorithme diviser pour régner : entrées : $P \subseteq E$ et deux tableaux X (respectivement y) contenant les points de P triés par ordre croissant sur les abscisses (respectivement sur les ordonnées).

On ne fait pas ce tri sur les tableaux à chaque récursion de l'algorithme.

Principe : si $|P| \geq 3$ on applique la méthode naïve. Sinon :

— Diviser : trouver une droite verticale l telle que l sépare P en deux sous-ensemble de même cardinal. On construit alors les nouveaux tableau qui sont triés. Méthode de la droite de balayage.

— Régner : calcul récursif de δ_G et δ_D. Puis on retourne le min des deux δ.

— Combiner : soit la réponse est δ soit la solution se trouve dans les deux moitiés. dessin

Si ces points existent ils sont dans une bande 2δ centrée en l. On construit Y' contenant l'ensemble des points dans cet borne rangée par ordre croissant. Pour chacun des points de Y', on compare δ à sa distance parmi les 7 points suivants dans Y'. Si on trouve une distance plus petite on la garde comme nouveau δ.

— On retourne alors les points fabriquant δ.

Complexité : $O(n \log n)$ Application du master théorème.

Application : contrôle de trafic (aérien, maritime) détection de collision.

Recherche d'une intersection entre deux segments [2, p.940]

Objectif : Recherche d'un élément lorsque $E = \{[(x_i, y_i), (x'_i, y'_i)]\}$ pour $i \in \{1, \dots, n\}$ est un ensemble de segments dans le plan.

Hypothèses : E ne contient pas de segment vertical et trois segments ne s'intersectent pas en un même point.

L'algorithme se base sur le principe de balayage selon x qui nous permet de trier les segments et faire une détection de l'intersection (l'ordre des segments s'inversent).

Écrire l'algorithme + dessin

Implémentation : arbre rouge-noir

Complexité pour trouver une intersection : $O(n \log n)$ dans le pire cas.
Complexité pour trouver toutes les intersections : $\Omega(n^2)$ dans le pire cas.
Les hypothèses peuvent être enlevées mais le traitement de ces cas limites complexifie grandement l'algorithme et sa preuve de validité.

Références

[1] D. Beauquier, J. Berstel, and P. Chrétienne. *Eléments d'algorithmique*. Manuels informatiques Masson. Masson, 1992.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Algorithmique, 3ème édition*. Dunod, 2010.

[3] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*. Springer, 2008.

[4] C. Froidevaux, M.C. Gaudel, and M. Soria. *Types de données et algorithmes*. McGraw-Hill, 1990.

[5] D. Knuth. *The Art of Computer Programming, vol 3*. Addison-Wesley, 1998.

[6] F. P. Preparata and M. I. Shamos. *Computational Geometry : an introduction*. Springer-Verlag, 1985.