

Reconnaissance de motifs textuels

Pour un problème donné, il n'y a pas d'algorithme universel. Certains algorithmes ne sont efficaces que dans un cas de figure particulier.

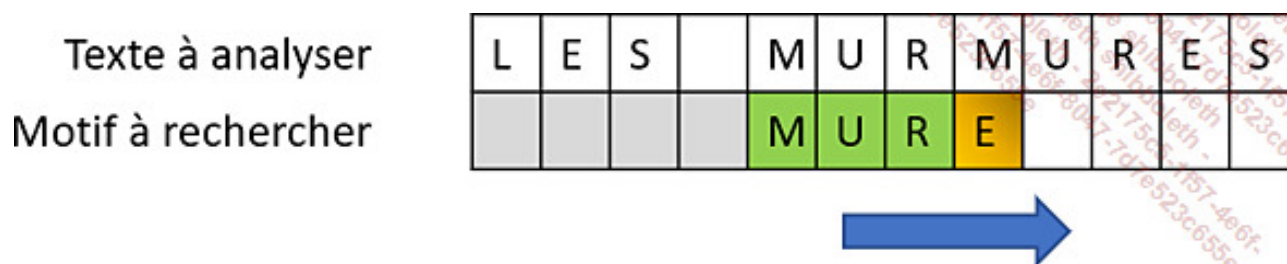
Nous nous intéressons ici à la recherche de motifs textuels, c'est-à-dire la recherche d'une sous-chaîne dans une chaîne et nous allons présenter plusieurs formes d'algorithmes.

La STL, tout comme boost, propose bien entendu des fonctions pour réaliser cela, mais la logique employée pour l'implémentation n'est pas indiquée dans le fichier d'en-tête. Il est utile de connaître quelques algorithmes de référence pour les appliquer à bon escient.

1. Approche directe

L'approche naïve consiste à comparer caractère par caractère le motif recherché avec le texte d'entrée. Si toutes les comparaisons réussissent du premier au dernier caractère du motif, on dit que le motif est épuisé et la recherche est terminée.

Si la comparaison diffère avant d'arriver à la fin du motif, il est inutile de poursuivre les comparaisons pour les caractères suivants. On décale le motif d'un caractère et on recommence la comparaison.



L'implémentation C++ de cet algorithme ne présente aucune difficulté. On doit vérifier les caractères un par un sans dépasser la taille du motif ni celle du texte à analyser :

```
int recherche_motif_simple(string s, string m)
{
    int pos = 0;
    int p = 0;
```

```

while (pos < s.length())
{
    if (s.at(pos) == m.at(0))
    {
        // premier caractère identique
        p = 1;

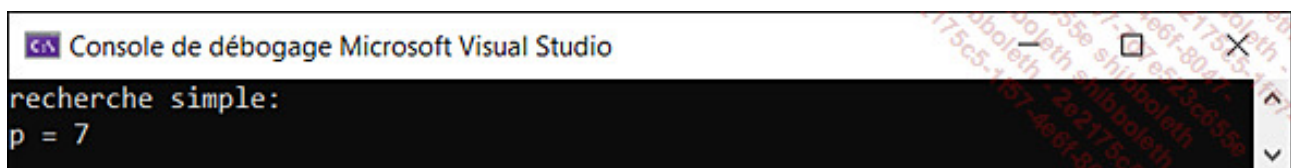
        // compare un par un les prochains caractères du motif
        while (
            p + pos < s.length() &&
            p < m.length() &&
            s.at(pos + p) == m.at(p)
        )
            p++;
        if (p == m.length())
            return pos;
    }

    // on décale le motif d'une position
    pos++;
}
return -1;
}

int main()
{
    string texte = "les murmures du vent dans les arbres";
    string motif = "mures";

    cout << "recherche simple:\n";
    int p = recherche_motif_simple(texte, motif);
    cout << "p = " << p << endl;
}

```



Cet algorithme pose des problèmes de performance, puisque dans le meilleur des cas, la

complexité est $O(n+m)$, si le premier caractère de m est rare dans le texte, et dans le pire des cas $O(n.m)$.

Voici quelques idées d'amélioration :

- ˆ Si un caractère lu ne se trouve pas dans le motif, on peut sauter le motif au complet. S'il se trouve plus loin dans le patron, on peut en sauter une partie (algorithme Boyer Moore).
- ˆ On pourrait repartir directement après le dernier caractère comparé dans le texte (algorithme Knuth Morris Pratt) puisqu'on a déjà lu le texte qui le précède.
- ˆ En associant un code de hachage aux m derniers caractères lus, on peut, en une comparaison avec le code de hachage du motif, déterminer s'il y a égalité (Rubin Karp).

2. Lecture avec déplacement : l'algorithme Boyer Moore

Boyer Moore est adapté aux motifs non répétitifs. Il utilise l'idée suivante : si un caractère ou une portion de texte ne se trouve nulle part dans le motif, on peut sauter toute la longueur du motif. Si le caractère lu se trouve dans le motif, on peut sauter au plus proche.

L'algorithme procède à reculons pour la comparaison :

- ˆ On compare les caractères un par un.
- ˆ S'il y a une différence, on saute au prochain endroit du motif qui contient ce caractère.

Prenons l'exemple d'un motif CACGGACCT constitué des caractères A, C, G, T et d'une longueur de 9 caractères.

On construit une table de saut à partir de la première position de chaque caractère en partant de la fin :

A	C	G	T	Autre caractère
3	1	4	9	9

Remarquons que la dernière lettre du motif a pour valeur la taille de ce motif. Étant le dernier caractère du motif, si la comparaison est négative, il faut repartir avec une fenêtre complète.

Pour le motif CTCGGACCT, nous aurons plutôt :

A	C	G	T	Autre caractère
3	1	4	9	9

L'implémentation fonctionne en deux temps, la construction de la table de saut puis la recherche proprement dite.

```
int recherche_boyer_moore(string s, string m)
{
    unique_ptr<int> saut (new int[256]);
    int i, j;

    // calculer la table des sauts
    for (j = 0; j < 256; j++)
        saut.get()[j] = m.length();

    for (j = 0; j < m.length() - 1; j++)
        saut.get()[m.at(j)] = m.length() - 1 - j;

    // rechercher
    int x = 0;
```

```

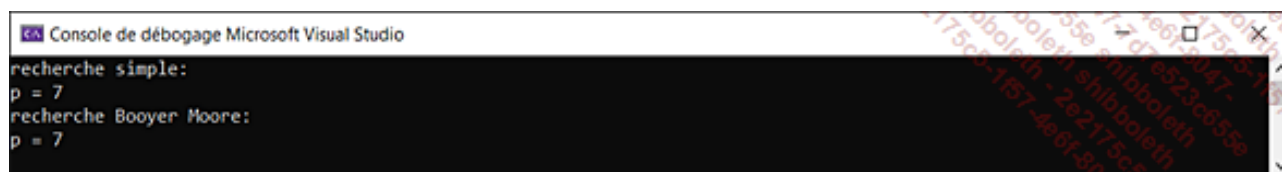
while (x <= s.length() - m.length())
{
    j = m.length() - 1;
    while ((j >= 0) && m.at(j) == s.at(x + j))
        j--;

    if (j == -1)
        return x;

    x = x + saut.get()[s.at(x + m.length() - 1)];
}
return -1;
}

```

L'exécution donne naturellement le même résultat que la recherche simple, mais avec beaucoup moins de comparaisons :



3. Méthode pour les motifs auto répétitifs : l'algorithme KMP

La méthode Knuth Morris Pratt s'appuie sur une portion de texte rejetée à cause d'une différence ; celle-ci peut tout de même contenir une plus petite portion de la chaîne qui sera complétée avec les prochains caractères.

Cela n'est valable que si le motif recherché est auto répétitif (son début se retrouve plus loin dans le motif). Sans être la majorité des situations, le temps de recherche est considérablement amélioré par rapport aux approches classiques. De plus, cette méthode évite de revenir constamment en arrière.

Sachant où on est rendu avant de détecter une différence, on détermine quelle portion au début du motif pourrait néanmoins être couverte par les caractères déjà lus. On obtient

alors une table appelée **suite** qui indique jusqu'où le motif est couvert avec ce qui a déjà été lu ; autrement, on oublie les caractères lus et on recommence la comparaison au début avec le caractère qui a causé la différence.

Étudions quelques exemples issus du monde de la génétique. L'alphabet limité à 4 caractères A, C, G, T favorise les répétitions de séquences au sein du motif.

~ Premier exemple :

Texte	AGCCTATCACATT
Motif	AGCGG
	1234
Suite	AGCGC

Erreur en position n°4 on décale de 3 positions.

~ Deuxième exemple :

Texte	AGCCTAGCACATT
Motif	AGCCTAT
	1234567
Suite	AGCCTAT

Erreur en position n°7, on remarque que T6 = M1, on décale M1 sur T6.

~ Troisième exemple :

Texte	AGCCGAG G TCATTAGTAAAAA
Motif	AGCCGAG C AGGC
	1 2 3 4 5 6 7 8
Suite	AGCCGAGCAGGC

Erreur en position n°8, on remarque que $T_{6,7} = M_{1,2}$, on décale M1 sur T6.

Comme pour Boyer Moore, le code fonctionne en deux temps avec la détermination de la table **suite** :

```
int recherche_kmp(string s, string m)
{
    int i, j;
    unique_ptr<int> suite(new int[m.length() + 1]);

    // préparer la table
    int k = 0;

    for (int q = 1; q < m.length(); q++)
    {
        while ((k > 0) && m.at(k) != m.at(q))
            k = suite.get()[k - 1];

        if (m.at(k) == m.at(q))
            k++;
        suite.get()[q] = k;
    }

    // recherche proprement dite
    int q = 0;
```

```

for (i = 0; i < s.length(); i++)
{
    while ((q > 0) && (m.at(q) != s.at(i)))
        q = suite.get()[q - 1];
    if (m.at(q) == s.at(i))
        q++;
    if (q == m.length())
    {
        q = suite.get()[q - 1];
        return (i - m.length() + 1);
    }
}
return -1;
}

```

La valeur de `suite[k]` est la plus petite valeur $s > 0$ pour laquelle $m[0, k-s]$ égale $m[s, k]$.

Voici le résultat de l'exécution avec les paramètres :

```

string texte = "les murs murmures du vent dans les arbres";
string motif = "murmures";

```

```

Microsoft Visual Studio
recherche simple:
p = 9
recherche Boyer Moore:
p = 9
recherche KMP:
p = 9

```

4. Méthode pour des motifs variables

Dans le cas de motifs variables, les trois algorithmes précédents sont inopérants. Il faut utiliser des automates à état fini pour traiter ces recherches. Si l'implémentation de ces

automates est d'une complexité modérée, il en est autre de leur paramétrage.

Heureusement, la bibliothèque STL propose un modèle d'expressions régulières très complet et très performant. Voici à titre d'exemple comment rechercher dans une chaîne tous les mots formés par `mur` ou une répétition de `mur`, c'est-à-dire `mur`, `murmur`, `murmurmur`, etc.

```
#include <iostream>
#include <ostream>
#include <string>
#include <regex>

using namespace std;
int main()
{
    string texte = "les murs murmures du vent dans les arbres";
    string motif = "murmures";
    int p;

    // recherche d'une correspondance
    const std::regex txt_regex("(mur)+");
    cout << "match : " << regex_search(texte, txt_regex) << '\n';

    // recherche de toutes les correspondances
    smatch pieces_match;
    string::const_iterator searchStart(texte.cbegin());

    while (regex_search(searchStart, texte.cend(), pieces_match,
txt_regex)) {

        // détail de la correspondance
        for (size_t i = 0; i < pieces_match.size(); i++) {
            ssub_match sub_match = pieces_match[i];
            string piece = sub_match.str();
            auto p = sub_match.first-texte.begin();
            cout << "\tcorrespondance #" << i << " " << p << " :
" << piece << '\n';
        }
        cout << "\n";
    }
}
```

```
// on poursuivra après la fin de cette correspondance  
searchStart = pieces_match.suffix().first;  
}  
}
```



The screenshot shows the 'Console de débogage Microsoft Visual Studio' window. It displays the output of a program, showing a match result and corresponding indices for 'mur' and 'murmur'.

```
match : 1  
correspondance #0 4 : mur  
correspondance #1 4 : mur  
  
correspondance #0 9 : murmur  
correspondance #1 12 : mur
```