Université de NANTES
Faculté des Sciences et des Techniques
Licence Informatique

Nantes le 03/05/2003

TP UL5
Reconnaissance de modèles

Abdeslam MOKRANI
Jérémy VALAYER
LIN Groupe 2

Dans ce projet nous allons réaliser un outil qui permettra de rechercher des mots clés dans un fichier texte donné. Le programme final affichera, pour chaque mot clé trouvé, le numéro de ligne dans laquelle il se trouve et l'indice du premier caractère du mot clé dans la ligne.

Le nom du fichier et les différents mots clés seront fournis comme arguments à notre programme. Celui ci permettra de gérer les erreurs éventuelles lors de lecture du fichier. Il fournira une option pour afficher les différentes étapes de calculs effectués. Il permettra, en outre, de donner un aide complet sur son utilisation.

Une version triviale serait d'implemter un algorithme qui testerait si chaque caractère lu dans le fichier s'agit de début d'un mot clé. Cette version serait très coûteuse en temps d'exécution. Car pour chaque caractère du fichier, il faut parcourir toute la liste des mots clés, et pour chaque mot clé, il faut tester l'égalité de celui-ci avec les caractères suivant le caractère lu (recherche non linéaire dans le fichier).

L'objectif est de réaliser un algorithme qui effectuera une recherche en temps linéaire. La lecture du fichier se fera caractère par caractère avec un seul traitement pour chaque caractère lu. La complexité du code à exécuter à la lecture d'un caractère sera la même, quel que soit le nombre et le format des mots clés ou le format des données du fichier à traiter. Après initialisait de notre programme, la longueur du fichier à traiter sera l'élément essentiel influençant le temps d'exécution de la recherche.

Pour présenter notre travail, nous allons détailler, étape par étape, les différents algorithmes servant à la réalisation du programme final, dont la plupart est déjà donnée en énoncé. Une traduction en langage C sera réalisée pour chaque algorithme. Des sorties d'écran résultant de jeux de test seront présentées et analysées au fur et à mesure de l'implemtation de notre programme. Durant cela, nous allons s'appuyer sur les principes théoriques vus en cours et établir un rapprochement avec les différents algorithmes étudiés.

SOMMAIRE

1. Initialisation de la recherche

Avant le lancement de la recherche, il est important de réaliser un traitement sur les mots clés à rechercher, afin d'optimiser l'algorithme de recherche qui doit s'adapter aux fichiers volumineux. La méthode qui sera utilisée est la mise en œuvre d'un transducteur déterministe fini en utilisant les mots clés, où les entrées sont les caractères lus, et les sorties sont les mots clés reconnus et leurs positions dans le fichier. La lecture d'un seul caractère déclenche une transition d'état qui est déterminé par la valeur du caractère et l'état courant. Un état sera considéré comme "final" si la disposition des derniers caractères lus correspond à un ou plusieurs mots clés (plusieurs mots clés s'ils sont imbriqués, par exemple, les mots clés `"c"`, `"bc"` et `"abc"` sont reconnus à la lecture du dernier caractère de la chaîne de caractères `"abc"`). Dans ce cas, la sotie de l'état est les mots clé reconnus avec leurs positions. Pour les autres états, la sortie est vide (aucun mot clé reconnu après la lecture d'un caractère). Ce transducteur devrait lire tout texte contenant des caractères ASCII. Du point de vue théorique, cela implique que tous ses états doivent être finaux. Néanmoins, un mot clé `w` est reconnu si et seulement si le texte appartient au langage `(x+w)*` où `x` corseront à n'importe quel caractère ASCII.

Si un tel transducteur est réalisé, notre objectif sera atteint. En effet, après la lecture d'un caractère et suivant l'état courant de notre transducteur, le traitement à effectuer n'est que la transition à un nouvel état (un seul état grâce au déterminisme du transducteur) et l'affichage de la sortie de cette transition (si elle n'est pas vide). Nous pouvons alors traiter le prochain caractère du fichier.

La réalisation de ce transducteur s'effectuera sur trois étapes. Dans un premier temps, la base du transducteur est construite, tous les états sont alors calculés et liés entre eux par des transitions. Cependant, d'autres transitions resteront à établir. Cela se fera après le calcul d'une table appelée l'échec des états, qui permet de résoudre le problème de chevauchement de mots clés (lorsque le préfixe d'un mot clé est déjà lu pendant la reconnaissance d'une autre occurrence d'un mot clé). Enfin, il restera à compléter le transducteur avec les transitions nécessaires déterminées en partie à partir de l'échec calculé.

1.1. Construction de la base

L'algorithme `construire_base` présenté en énoncé montre comment réaliser cette étape. Nous avons choisi d'implemter cet algorithme en langage C en utilisant un tableau pour représenter les transitions du transducteur. Deux fonctions ont été définis dans cette étape. La fonction `init()` calcul le nombre maximal d'états (`borne`), qui correspond au nombre des caractères de tous les mots clés plus un (un autre état est réservé pour tous les autres caractères). Les caractères distincts faisant partie des mots clés sont extraits dans un tableau `caracteres[]` et leur nombre est calculé (`nbCara`), cela permet d'allouer dynamiquement la table de transitions qui est un tableau de `borne` × `nbCara` états. D'autres variables sont initialisées dans le corps de cette fonction tel que le tableau à deux dimensions `resultat` qui détermine les mots clés reconnus pour un état donné. Voir la fonction `init()` en annexe B page 15.

La fonction `construire_base()` permet de calculer le nombre réel des états et de remplir partiellement la table de transitions et le tableau `resultat` (voir annexe B page 16).

Fonctionnement

Cette fonction calcule pour chaque mot clé un automate permettant de le reconnaître. Si plusieurs mots clés ont les mêmes préfixes, alors leurs automates sont unis de manière à parcourir les mêmes états pour un préfixe commun. Tous les automates calculés sont unis pour avoir le même état initial 0. Une représentation graphique d'un tel automate ressemble alors à un arbre (voir l'exemple donné ci-dessous). Les états finaux de cet automate (feuilles de l'arbre) correspondent aux états avec une sortie non nulle dans le transducteur. Chaque sortie non nulle lors d'une transition à un état est définie par le mot clé reconnu par l'automate à cet état. Tous les mots clés reconnus sont ajoutés dans le tableau `resultat`.

Exemple

Ci-dessous un affichage produit par notre programme après l'exécution de la fonction `construire_base()` avec les mots clés `"NI"`, `"RENE"`, `"REIN"` et `"IRENE":`

```
bash-2.04$ ./recherche -v NI RENE REIN IRENE

borne=15
nb. cara. dist=4
long. maxi.=5
caracteres="NIRE"

BASE :
======

TRANSITION :
------------
       N   I   R   E

   0   1   3   2   0
   1   0   4   0   0
   2   0   0   0   5
   3   0   0   6   0
   4   0   0   0   0
   5   7   8   0   0
   6   0   0   0   9
   7   0   0   0  10
   8  11   0   0   0
   9  12   0   0   0
  10   0   0   0   0
  11   0   0   0   0
  12   0   0   0  13
  13   0   0   0   0
  14   0   0   0   0
  15   0   0   0   0

RESULTAT :
----------
  Etat[1] :
  Etat[2] :
  Etat[3] :
  Etat[4] : "NI"
  Etat[5] :
  Etat[6] :
  Etat[7] :
  Etat[8] :
  Etat[9] :
  Etat[10] : "RENE"
  Etat[11] : "REIN"
  Etat[12] :
  Etat[13] : "IRENE"
  Etat[14] :
  Etat[15] :
```

Le programme affiche le nombre de tous les caractères (15), le nombre des caractères distincts (4), la longueur du plus long mot clé (5) et les caractères distincts `"NIRE"` . La table de transitions est affichée suivie du tableau `résultat` qui détermine les états finaux avec les mots clés reconnus.

A partir de ces données affichées, on peut tracer une représentation graphique de l'automate calculé, voir la figure 1.

L'automate de la figure 1 ne représente pas exactement la table de transitions affichée. En effet, il faut ajouter une transition pour chaque autre caractère non pris en compte à chaque état vers l'état initial 0. Nous devons considérer qu'à un état donné, si on lit un caractère n'appartenant pas à l'ensemble des caractères des mots clés, alors on transite vers l'état initial 0, ce qui augmentera considérablement la taille de la table de transitions. C'est la raison pour laquelle nous n'avons pas considéré l'ensemble des caractères ASCII (cependant, un simple test permettra d'effectuer cette transition lors de la lecture du fichier).

Le transducteur calculé jusqu'ici ne peut pas retrouver quelques occurrences de mots clés dans un texte donné comme le montre l'exemple suivant :

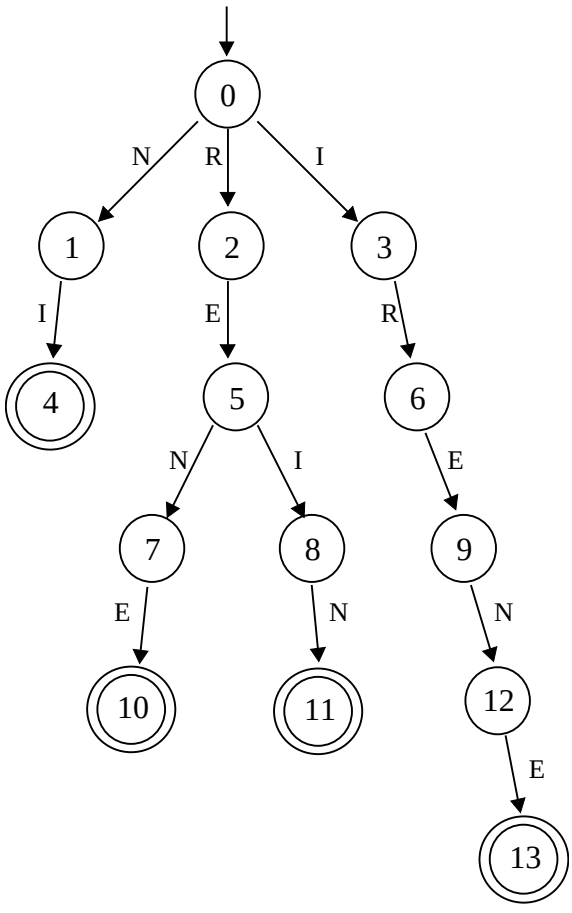
Mots clés à rechercher : "NI", "RENE", "REIN", "IRENE"
Texte : "abNIcdNIRENEefIRENE"
Calcul :

$$0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{N} 1 \xrightarrow{I} 4\ (\text{NI reconnu}) \xrightarrow{c} 0 \xrightarrow{d} 0 \xrightarrow{N} 1 \xrightarrow{I} 4\ (\text{NI reconnu}) \xrightarrow{R} 0 \xrightarrow{E}$$
$$0 \xrightarrow{N} 0 \xrightarrow{E} 0 \xrightarrow{e} 0 \xrightarrow{f} 0$$

Dans cet exemple, les deux premières occurrences du mot "NI" sont retrouvées, la première occurrence du mot "IRENE" n'est pas retrouvée. Cela s'explique du fait que le préfixe "I" du mot "IRENE" constitue au même temps un suffixe au mot "NI" reconnu précédemment. D'où l'importance d'établir une fonction pour le calcul de l'échec des états, qui va être pris en compte lors du complètement du transducteur. L'occurrence "RENE" suivant le mot "NI" n'est pas reconnu, car aucune transition ne permet le passage à l'état 2 à partir de l'état 4 en lisant le caractère "R".

Nous allons voir maintenant comment tenir compte du chevauchement de mots clés ou de leurs préfixes en calculant l'échec des états.

Figure 1 ***automate de reconnaissance de mots clés***



1.2. Construction de l'échec des états

L'échec des états correspond à un tableau qui à chaque état en indice associe un autre état. Un état `echec[e]` autre que l'état initial 0 est associé si l'état indice `e` est différent de 0 (au moins un mot clé ou son préfixe est lu), et pendant la lecture amenant à l'état `e`, on aurait pu commencer à lire un autre mot clé `w2`. L'état `echec[e]` est alors l'état obtenu si le préfixe du mot clé w2 a été lu. L'intérêt de ce tableau est d'indiquer quels sont les mots clés qu'on a lus ou est entrain de lire. Cela nous évite de faire des allers retours dans le texte pour reconnaître tous les mots clés possibles.

On ne transite sur l'état échec de l'état actuel que si le mot clé qui a conduit à l'état actuel est complètement lu ou le caractère suivant (dans le texte) ne permet pas de reconnaître le mot clé (échec de lecture).

Nous avons implemté l'algorithme `construire_echec` en langage C dans la fonction `construire_echec()` présentée en annexe B page 16. Celle-ci calcule l'échec des états dans le tableau `echec[]` convenablement alloué pour le nombre d'états déjà calculé.

Fonctionnement

Pour chaque état `e` différent de 0, l'état échec `echec[e]` associé est l'état qu'on aurait obtenu à la lecture du plus long suffixe du texte déjà lu (certainement faisant parti d'un mot clé) qui correspond à un préfixe d'un mot clé.

Exemple

Ci-dessous l'affichage du tableau `echec[]` produit par le programme après l'exécution de la fonction `construire_echec()` avec les mots clés `"NI"`, `"RENE"`, `"REIN"` et `"IRENE":`

```
bash-2.04$ ./recherche -v NI RENE REIN IRENE

ECHEC :
-------
   0    1    2    3    4    5    6    7    8    9   10   11   12   13
   0    0    0    0    3    0    2    1    3    5    0    1    7   10
```

Si nous regardons la visualisation graphique de la figure 1, nous remarquons qu'à l'état 4, le transducteur lit le mot `"NI"`. Le caractère `"I"` constitue au même temps un suffixe au mot `"NI"` qu'on vient de lire et un préfixe au mot clé `"IRENE"`. Il est le plus long mot de ce type. A la lecture du mot `"IRENE"`, on arrive à l'état 3 après lecture du `"I"` (voir figure 1). L'échec de l'état 4 est donc bien 3, comme le montre l'affichage précédent. Il est de même avec les états 6 et 2 (suffixe `"R"`), 9 et 5 (suffixe `"RE"`) etc..

1.3. Complètement

Cette étape permet d'ajouter toutes les transitions nécessaires pour reconnaître toutes les occurrences de mots clés dans un texte donné quel que soit leur emplacement (ou modifier quelques transitions vers l'état 0 si on considère le tableau de transitions).

Les transitions manquantes sont liées à deux types de comportement :

- Le premier concerne, comme on vient de le voir, l'échec des états.
- Le second consiste au fait qu'on ne doit pas avoir des transitions sans entrée dans notre transducteur (entrée = ε). Car ce concept théorique n'est pas applicable dans notre situation où l'on doit absolument lire un caractère à chaque transition. Or jusqu'ici, les transitions sans entrée sont considérées comment étant des

transitions valables pour toute entrée. Cela arrive, par exemple, lorsque la fin d'un mot clé est reconnu et que l'on doit transiter sur l'état 0.

Nous pouvons remarque que seules les transitions vers l'état 0 sont concernées. En effet, toutes les autres transitions construites (algorithme `construire_base`) ont une entrée non vide et en cas d'échec ou de fin de lecture d'un mot clé, l'état suivant est toujours l'état 0.

La solution est semblable à l'algorithme de suppression de transitions vides vu en cours. Dans notre cas, nous ne traitons que les états différents de 0 (aucune partie d'un mot clé n'est lue à l'état 0, donc pas de transition à entrée vide à partir de cet état). La fonction `compléter()` (annexe B page 16) implemte l'algorithme concerné.

Fonctionnement

Pour chaque état `e` différent de 0 et pour tout caractère `c` appartenant à un mot clé, si la transition de l'état `e` en lisant le caractère `c` amène à l'état 0, alors on la modifie par une transition qui a comme :

- état de départ : l'état `e`
- entrée : le caractère `c`
- état d'arrivée : l'état sur lequel on transite à partir de l'état `echec[e]` (0 si aucun état d'échec) en lisant le caractère `c` (les deux types de comportement sont alors gérés en même temps)

Il arrive parfois de reconnaître plusieurs mots clé à un même état `e`. Cela survient quand des suffixes d' occurrences de mots clés se coïncident dans le texte (par exemple, on reconnaît les mots clés `"c"`, `"bc"` et `"abc"` dans le texte `"abc"` à l'état d'arrivée en lisant le caractère `"b"`). Les mots clés reconnus sont alors ceux contenant les suffixes concernés. Dans tous les états de ce type, nous avons toujours un état d'échec non nul associée. Il faut ajouter tous les mots clés reconnus au résultat de l'état `e` dans le tableau `resultat[]`. Ces mots clé sont ceux reconnus par l'echec de l'état `e`.

Exemple

Voici ce que l'on obtient après l'exécution de la fonction `completer()` avec les mots clé `"NI"`, `"RENE"`, `"REIN"` et `"IRENE":`

```
bash-2.04$ ./recherche -v NI RENE REIN IRENE

COMPLETEMENT :
==============

TRANSITION :
------------
       N   I   R   E

   0   1   3   2   0
   1   1   4   2   0
   2   1   3   2   5
   3   1   3   6   0
   4   1   3   6   0
   5   7   8   2   0
   6   1   3   2   9
   7   1   4   2  10
   8  11   3   6   0
   9  12   8   2   0
  10   1   3   2   0
  11   1   4   2   0
  12   1   4   2  13
  13   1   3   2   0
  14   0   0   0   0
  15   0   0   0   0

RESULTAT :
----------
  Etat[1] :
  Etat[2] :
```

```
Etat[3] :
Etat[4] : "NI"
Etat[5] :
Etat[6] :
Etat[7] :
Etat[8] :
Etat[9] :
Etat[10] : "RENE"
Etat[11] : "REIN"
Etat[12] :
Etat[13] : "IRENE" "RENE"
Etat[14] :
Etat[15] :
```

D'après cet affichage, nous remarquons que beaucoup de transitions sont ajoutées et qu'en reconnaissant le mot clé "IRENE", le mot clé "RENE" est aussi reconnu.

Le transducteur est maintenant bien initialisé, il reste à l'utiliser dans un algorithme qui permettra de l'interroger afin de réaliser une recherche dans un fichier.

2. Recherche de mots clés dans un fichier

Pour rechercher des occurrences de mots clé dans un fichier, il faut au moins un transducteur initialisé avec les mots clés, et le fichier. L'algorithme suivant permet de réaliser une telle recherche :

```
Procedure recherche(
    {paramètres}
    fic : Fichier Texte,

    transition : Tableau[1..borne, 1..nbCara],          } Transducteur
    resultat : Tableau[1..nbEtats, 1..nbMotsCles] );    }

Var
    {variables locales}
    e       : Un etat;
    ligne   : entier; {numéro de ligne}
    pos     : entier; {indice dans la ligne}
    cara    : caractère; {le caractère à lire}

Début
    e ← 0; {état initiale}
    ligne ← 1;
    pos ← 0;
    Tant que (NON FinFichier(fic)) faire
          cara ← lireCaractère(fic);
          Si cara = caractère de fin de ligne alors
             écrire(e,' -\n-> 0 ');
             e ← 0;
             pos ← 0;
             ligne ← ligne + 1;
          Sinon
          début
             écrire(e, ' -', cara, '-> ');
             pos ← pos + 1;
             e ← transition[e, cara];
             si resultat[e] non vide alors
             début
```

```
            écrire(e,'final, mots clés reconnus :');
            pour i<-1 à nombre(resultat[e]) faire
                écrire('ligne: ', ligne,
                       ' indice: ', pos-longeur(resultat[e, i]),
                       'mot clé : resultat[e, i]);
            Fin pour
        Fin si
      Fin si
   Fin Tant que
   écrire(e,'-Fin Fichier-> 0');
Fin
```

Cet algorithme est implemté en langage C dans la fonction `traite_fichier()` (annexe B page 17). Cette fonction prend en comptes tous les autres caractères ASCII et affecte à l'état courant l'état initial 0 sans utiliser le transducteur. Elle permet d'afficher (si la variable `v` est égale à 1) l'ensemble de transitions effectuées pendant la recherche et indique les mots clés retrouvés ainsi que leurs positions dans le fichier texte. Ci-dessous un exemple d'affichage généré par cette fonction :

```
bash-2.04$ echo "abNIcdNIRENEhREINInb
> RENEdfdgNINIfdfgRENE
> dfgNI
> df"|./recherche -v NI RENE REIN IRENE

RECHERCHE :
===========
0 -'a'-> 0 -'b'-> 0 -'N'-> 1 -'I'-> 4 final
[1, 3] : "NI"
4 -'c'-> 0 -'d'-> 0 -'N'-> 1 -'I'-> 4 final
[1, 7] : "NI"
4 -'R'-> 6 -'E'-> 9 -'N'-> 12 -'E'-> 13 final
[1, 8] : "IRENE"
[1, 9] : "RENE"
13 -'h'-> 0 -'R'-> 2 -'E'-> 5 -'I'-> 8 -'N'-> 11 final
[1, 14] : "REIN"
11 -'I'-> 4 final
[1, 17] : "NI"
4 -'n'-> 0 -'b'-> 0 -'\n'-> 0
0 -'R'-> 2 -'E'-> 5 -'N'-> 7 -'E'-> 10 final
[2, 1] : "RENE"
10 -'d'-> 0 -'f'-> 0 -'d'-> 0 -'g'-> 0 -'N'-> 1 -'I'-> 4 final
[2, 9] : "NI"
4 -'N'-> 1 -'I'-> 4 final
[2, 11] : "NI"
4 -'f'-> 0 -'d'-> 0 -'f'-> 0 -'g'-> 0 -'R'-> 2 -'E'-> 5 -'N'-> 7 -'E'-> 10 final
[2, 17] : "RENE"
10 -'\n'-> 0
0 -'d'-> 0 -'f'-> 0 -'g'-> 0 -'N'-> 1 -'I'-> 4 final
[3, 4] : "NI"
4 -'\n'-> 0
0 -'d'-> 0 -'f'-> 0 -'\n'-> 0
0 -'EOF'-> 0
```

Nous remarquons que toutes les occurrences des mots clés passés en paramètre à notre programme ont été retrouvées dans le texte redirigé. Nous pouvons suivre le déroulement de la recherche et vérifier en regardant la table de transitions affichée précédemment pour les mêmes mots clés.

Avant l'initialisation et l'appel de la fonction de recherche, le programme principale doit gérer l'entrée des données par l'utilisateur et les erreurs qui peuvent survenir. C'est ce que nous allons expliquer dans la prochaine section.

3. Le programme principal

Nous avons adapté notre programme pour qu'il soit utilisable en tant que commande. Des options peuvent être indiquées sur la ligne de commande, ceci est géré grâce à l'argument de type `char **` qui, passé à la fonction `main()`, permet d'accéder aux arguments de la commande. L'option `-h` permet d'afficher un aide sur l'utilisation du programme, notamment, les différentes options possibles. Voici ce qui est affiché si cette option est mise.

```
   bash-2.04$ ./recherche -h
   ./recherche.exe permet de rechercher des mots clés dans un fichier texte ou sur l'entrée
standard

   Syntaxe :
   ---------
     ./recherche.exe [-h] [-v] [-f fichier] mot_clé1 mot_clé2 ..

   Options :
   ---------
     -h permet d'affichier cet aide, les autres arguments éventuels sont ignorés
     -v permet d'afficher toutes les étapes de calculs (initialisation et
        recherche de mots clés)
     -f permet de traiter un fichier au lieu de lire les données à partir de
         l'entrée standard
       l'argument qui suit cette option doit être le nom du fichier à traiter

   Affichage :
   -----------
     Chaque mot clé trouvé est affiché sur une ligne avec la syntaxe
       [ligne, colonne] : "mot_clé"
     où
       ligne  : le numéro de la ligne où se trouve le mot clé
       colonne : l'indice où commence le mot clé dans la ligne
       mot_clé : le mot clé retrouvé

   Remarques :
   -----------
     - Toutes les options doivent être spécifiées par ordre, sinon, elles seront
       considérées comme des mots clés
     - Au moins un mot clé doit être donné
     - Si aucun mot clé n'est donné après une option, celle-ci sera considérée
       comme un mot clé
```

Le fichier dans lequel s'effectuera la recherche peur être donné sur la ligne de commande. Dans ce cas, le programme principale essaie de l'ouvrir. Si une erreur survient, alors la valeur de cette erreur est affichée, comme le montre l'exemple suivant :

```
bash-2.04$ ./find -f nexistepas a b c
nexistepas: No such file or directory
```

Si aucun fichier n'est indiqué sur la ligne de commande, alors le programme utilise l'entrée standard (clavier de l'utilisateur ou redirection).

Pour permettre aux autres commandes de manipuler le résultat fourni par le programme, l'affichage est restreint à la position de chaque mot clé ainsi que le mot clé lui même. Toutefois, nous pouvons suivre le déroulement de l'initialisation et de la recherche en spécifiant l'option `-v`.

Dans tout le programme, les variables principales sont globales. Cela nous évite de les faire passer en paramètre de chaque fonction. Les tableaux alloués dynamiquement sont tous libérés à la fin du programme principale et le fichier est fermé.

Le code de la fonction `main()` représentant le programme principal est en annexe B page 18.

4. Conclusion

Ce projet nous montre l'intérêt du fondement théorique de l'informatique, notamment sur le traitement des langages. En effet, cela amène des preuves mathématiques sur le bon fonctionnement de notre programme. Nous avons vu que la fonctionnalité voulue peut être réalisée d'une façon triviale, mais cela induit généralement une inefficacité des programmes de point de vue temps d'exécution.

Ce programme est un bon exemple pour expliquer le fonctionnement d'un compilateur d'un langage informatique. En effet, l'un et l'autre utilisent en entrée un texte, vérifient sa syntaxe à l'aide d'un automate puis génèrent une sotie suivant le texte entré.

5. Annexe A : jeux d’essais

Ci-dessous, une série de jeux d’essais illustrant l'utilisation du programme.

```
bash-2.04$
bash-2.04$ ./recherche -v bc c d
borne=4
nb. cara. dist=3
long. maxi.=2
caractere="bcd"

BASE :
======

TRANSITION :
------------
       b   c   d

   0   1   2   3
   1   0   4   0
   2   0   0   0
   3   0   0   0
   4   0   0   0

ECHEC :
-------
   0    1    2    3    4
   0    0    0    0    2

RESULTAT :
----------
  Etat[1] :
  Etat[2] : "c"
  Etat[3] : "d"
  Etat[4] : "bc"

COMPLETEMENT :
==============

TRANSITION :
------------
       b   c   d

   0   1   2   3
   1   1   4   3
   2   1   2   3
   3   1   2   3
   4   1   2   3

ECHEC :
-------
   0    1    2    3    4
   0    0    0    0    2

RESULTAT :
----------
  Etat[1] :
  Etat[2] : "c"
  Etat[3] : "d"
  Etat[4] : "bc" "c"

RECHERCHE :
===========
azerbcjdddc
0 -'a'-> 0 -'z'-> 0 -'e'-> 0 -'r'-> 0 -'b'-> 1 -'c'-> 4 final
[1, 5] : "bc"
[1, 6] : "c"
4 -'j'-> 0 -'d'-> 3 final
[1, 8] : "d"
3 -'d'-> 3 final
[1, 9] : "d"
3 -'d'-> 3 final
[1, 10] : "d"
3 -'c'-> 2 final
[1, 11] : "c"
2 -'\n'-> 0
```

```
abdcb
0 -'a'-> 0 -'b'-> 1 -'d'-> 3 final
[2, 3] : "d"
3 -'c'-> 2 final
[2, 4] : "c"
2 -'b'-> 1 -'\n'-> 0
0 -'EOF'-> 0
bash-2.04$
bash-2.04$
bash-2.04$
bash-2.04$ ./recherche bc c d
abdhbcbcbbcc
[1, 3] : "d"
[1, 5] : "bc"
[1, 6] : "c"
[1, 7] : "bc"
[1, 8] : "c"
[1, 10] : "bc"
[1, 11] : "c"
[1, 12] : "c"
dcbde
[2, 1] : "d"
[2, 2] : "c"
[2, 4] : "d"
d
[3, 1] : "d"
bash-2.04$
bash-2.04$
bash-2.04$
bash-2.04$ cat > dat.dat
abcbbbccdj
cdbccbcbddsf
sdfjkkjbcbosodijfiojcck
ddlkf
jk
bcdss
bash-2.04$ ./recherche -f dat.dat bc b d
[1, 2] : "bc"
[1, 3] : "c"
[1, 6] : "bc"
[1, 7] : "c"
[1, 8] : "c"
[1, 9] : "d"
[2, 1] : "c"
[2, 2] : "d"
[2, 3] : "bc"
[2, 4] : "c"
[2, 5] : "c"
[2, 6] : "bc"
[2, 7] : "c"
[2, 9] : "d"
[2, 10] : "d"
[3, 2] : "d"
[3, 8] : "bc"
[3, 9] : "c"
[3, 14] : "d"
[3, 21] : "c"
[3, 22] : "c"
[4, 1] : "d"
[4, 2] : "d"
[6, 1] : "bc"
[6, 2] : "c"
[6, 3] : "d"
bash-2.04$
bash-2.04$
bash-2.04$
bash-2.04$ ./recherche bc b d  < dat.dat
[1, 2] : "bc"
[1, 3] : "c"
[1, 6] : "bc"
[1, 7] : "c"
[1, 8] : "c"
[1, 9] : "d"
[2, 1] : "c"
[2, 2] : "d"
[2, 3] : "bc"
[2, 4] : "c"
[2, 5] : "c"
[2, 6] : "bc"
[2, 7] : "c"
```

```
[2, 9] : "d"
[2, 10] : "d"
[3, 2] : "d"
[3, 8] : "bc"
[3, 9] : "c"
[3, 14] : "d"
[3, 21] : "c"
[3, 22] : "c"
[4, 1] : "d"
[4, 2] : "d"
[6, 1] : "bc"
[6, 2] : "c"
[6, 3] : "d"
```

6. Annexe B : code source du programme

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

size_t m;         /* le nombre de mots clés*/
size_t borne;     /* le nombre de caractères de tous les mots clés*/
size_t nbCara;    /* le nombre de caractères distinctes de tous les mots clés*/
size_t maxLong;   /* le nombre de caractères du plus long mot clé*/
size_t nbEtats;   /* le nombre d'états*/

char **M;         /* tableau de tous les mots clés*/
int *longM;       /* tableau des longueurs des mots clés*/
char *caractere;  /* tableau des caractères distinctes de tous les mots clés*/
int **transition; /* tableau des transitions*/
int **resultat;   /* tableau des mots clés reconnus pour les états*/
int *echec;       /* tableau représentant la fonction d'échec des états*/
FILE *fic;        /* fichier*/

int v;            /* pour savoir s'il faut afficher le déroulement du calcul*/

void init()
/*   permet d'initialiser les variables :
     borne, nbCara, maxLong, longM, caracteres, transition et resultat
*/
{
  int i, j, temp;
  borne=0; nbCara=0; maxLong=0;
  longM=(int *) malloc(m*sizeof(int));
  for(i=0; i<m; i++)
  {
    temp=strlen(M[i]);
    longM[i]=temp;
    borne=borne+temp;
    if(temp > maxLong) maxLong=temp;
  }
  caractere=(char *) malloc((borne+1)*sizeof(char));
  caractere[0]='\0';
  for(i=0; i<m; i++)
  {
    strcat(caractere, M[i]);
  }
  transition=(int **) malloc((borne+1)*sizeof(int *));
  resultat=(int **) malloc((borne+1)*sizeof(int *));
  for(i=0; i<=borne; i++)
  {
    j=0;
    while(j<nbCara && caractere[i]!=caractere[j])
      ++j;
    if(j==nbCara)
    {
      ++nbCara;
      caractere[j]=caractere[i];
    }
    resultat[i]=(int *) malloc((m+1)*sizeof(int));
    resultat[i][0]=0;
  }
  --nbCara;
  for(i=0; i<=borne; i++)
  {
    transition[i]=(int *) malloc(nbCara*sizeof(int));
    for(j=0; j<nbCara; j++)
      transition[i][j]=0;
  }
}

int indCara(char cara)
/*  retourne l'indice du caractère 'cara' dans le tableau des caractères
    distincts 'caractere'
*/
{
  int ind=0;
  while(caractere[ind]!=cara && ind<nbCara)
    ++ind;
  return ind;
```

```
}

void inserer(int e, int indMotCle)
{
  int i=1;
  while(resultat[e][i]!=indMotCle && i<=resultat[e][0])
   ++i;
  if(i>resultat[e][0])
  {
    ++resultat[e][0];
    resultat[e][i]=indMotCle;
  }
}

void construire_base()
{
  int e, i, j, ind;
  int nouvelEtat;
  int *dernierEtat=(int *) malloc(m*sizeof(int));
  nbEtats=0;

  for(i=0; i<m; i++)
    dernierEtat[i]=0;

  for (j=0; j<maxLong; j++)
    for (i=0; i<m; i++)
      if (j<longM[i])
      {
        ind=indCara(M[i][j]);
        nouvelEtat=transition[dernierEtat[i]][ind];
        if (nouvelEtat==0)
        {
          ++nbEtats;
          nouvelEtat=nbEtats;
          transition[dernierEtat[i]][ind]=nouvelEtat;
        }
        if (j==longM[i]-1)
        {
          inserer(nouvelEtat, i);
        }
        dernierEtat[i]=nouvelEtat;
      }
  free(dernierEtat);
}

void construire_echec()
{
  int c, e, nouvelEtat, x, y=0;
  echec=(int *) malloc((nbEtats+1)*sizeof(int)); /* +1 pour l'état 0*/
  echec[0]=0;
  for(c=0; c<nbCara; c++)
    if(transition[0][c]!=0)
      echec[transition[0][c]]=0;
  for(e=1; e<=nbEtats; e++)
    for(c=0; c<nbCara; c++)
    {
      nouvelEtat=transition[e][c];
      if(nouvelEtat!=0)
      {
        x=e;
        do
        {
          x=echec[x];
          y=transition[x][c];
        }
        while(y==0 && x!=0);
        echec[nouvelEtat]=y;
      }
    }
}

void completer()
{
  int e, c, i;
  for(e=1; e<=nbEtats; e++)
   {
    for(c=0; c<nbCara; c++)
    if(transition[e][c] == 0)
      transition[e][c]=transition[echec[e]][c];

    for(i=1; i<=resultat[echec[e]][0]; i++)
```

```
      inserer(e, resultat[echec[e]][i]);
   }
}

void traite_fichier()
/*  recherche de tous les mots clés dans le fichier en affichant pour chaque
    mot clé trouvé le numéro de ligne où il se trouve et l'indice de son
    premier caractère dans cette ligne
*/
{
  int cara=0, ind, e=0, i, pos=0, ligne=1;
  while(cara>=0 && cara<=255)
  {
    cara=fgetc(fic);
    if(cara==10)
    {
      if (v) printf("%d -'\\n'-> 0\n", e);
      e=0; /* on suppose que le caractère de fin de ligne ne peut faire partie de
           l'ensemble des symboles*/
      pos=0;
      ++ligne;
    }
    else if(cara>=0 && cara<=255)
    {
      if (v) printf("%d -'%c'-> ", e, cara);
      ++pos;
      ind=indCara(cara);
      if (ind<nbCara)
      {
        e=transition[e][ind];
        if(resultat[e][0]>0)
        {
          if (v) printf("%d final\n", e);
          for(i=1; i<=resultat[e][0]; i++)
           printf("[%d, %d] : \"%s\"\n",ligne, pos-longM[resultat[e][i]]+1,
                  M[resultat[e][i]]);
        }
      }
      else
        e=0;
     }
  }
  if(v) printf("%d -'EOF'-> 0\n", e);
}

void affichage()
/*  affichage de la table de transitions, de l'état échec de chaque état
    et des mots clé reconnus à chaque état
*/
{
  int i, j;
  printf("\nTRANSITION :\n------------\n");
  printf("%4s", "");
  for(j=0; j<nbCara;j++)
  {
    printf("%4c", caractere[j]);
  }
  printf("\n\n");
  for(i=0; i<=borne; i++)
  {
    printf("%4d", i);
    for(j=0; j<nbCara;j++)
    {
      printf("%4d", transition[i][j]);
    }
    printf("\n");
  }
  printf("\nECHEC :\n-------\n");
  for(i=0; i<=nbEtats; i++)
    printf("%4d ", i);
  printf("\n");
  for(i=0; i<=nbEtats; i++)
    printf("%4d ", echec[i]);
  printf("\n\nRESULTAT :\n----------\n");
  for(i=1; i<=borne; i++)
  {
    printf("  Etat[%d] : ", i);
    for(j=1; j<=resultat[i][0]; j++)
      printf("\"%s\" ", M[resultat[i][j]]);
    printf("\n");
  }
```

```
}

main(int nb, char **args)
{
  int i;
  v=0;

  /* initialisations*/
  if(nb<=1)
    /* erreur, pas assez d'arguments*/
    fprintf(stderr, "Usage : %s [-h] [-v] [-f fichier] mot_clé1 mot_clé2 ..\n", args[0]);
  else if (strcmp(args[1], "-h")==0)
  {
    printf("%s permet de rechercher des mots clés dans un fichier\n", args[0]);
    printf("texte ou sur l'entrée standard\n\n");
    printf("Syntaxe :\n---------\n");
    printf("  %s [-h] [-v] [-f fichier] mot_clé1 mot_clé2 ..\n\n", args[0]);
    printf("Options :\n---------\n");
    printf("  -h permet d'affichier cet aide, les autres arguments éventuels\n");
    printf("     sont ignorés\n");
    printf("  -v permet d'afficher toutes les étapes de calculs \n");
    printf("(initialisation et recherche de mots clés)\n");
    printf("  -f permet de traiter un fichier au lieu de lire les données à"\n);
    printf("     partir de l'entrée standard\n");
    printf("     l'argument qui suit cette option doit être le nom du fichier\n");
    printf("     à traiter\n\n");
    printf("Affichage :\n-----------\n");
    printf("  Chaque mot clé trouvé est affiché sur une ligne avec la syntaxe\n");
    printf("    [ligne, colonne] : \"mot_clé\"\n  où\n");
    printf("    ligne  : le numéro de la ligne où se trouve le mot clé\n");
    printf("    colonne : l'indice où commence le mot clé dans la ligne\n");
    printf("    mot_clé : le mot clé retrouvé\n\n");
    printf("Remarques :\n-----------\n");
    printf("  - Toutes les options doivent être spécifiées par ordre, sinon,\n");
    printf("    elles seront considérées comme des mots clés\n");
    printf("  - Au moins un mot clé doit être donné\n");
    printf("  - Si aucun mot clé n'est donné après une option, celle-ci sera\n");
    printf("    considérée comme un mot clé\n");
  }
  else
  {
    if(nb>=3 && strcmp(args[1], "-v")==0) v=1;
    if(nb>=4+v && strcmp(args[1+v], "-f")==0)
     if((fic=fopen(args[2+v], "r"))!=NULL)
     {
       M=&args[3+v]; /* le tableau des mots cles = le tableau des argument*/
       m=nb-3-v;     /* nombre de mots cles = nombre d'arguments*/
     }
     else
     {
       perror(args[2+v]);
       return 1;
     }
    else
    {
      M=&args[1+v];
      m=nb-1-v;
      fic=stdin;
    }
    init();
    if(v) printf("\nborne=%d\nnb. cara. dist=%d\nlong. maxi.=%d\ncaractere=\"%s\"\n",
                 borne, nbCara, maxLong, caractere);
    construire_base();
    construire_echec();
    if(v)
    {
      printf("\nBASE :\n======\n");
      affichage();
    }
    completer();
    if(v)
    {
      printf("\nCOMPLETEMENT :\n==============\n");
      affichage();
      printf("\nRECHERCHE :\n===========\n");
    }

    traite_fichier();

    fclose(fic);
```

```
    /* liberation de la mémoire*/
    free(longM);
    free(caractere);
    for(i=0; i<borne; i++)
    {
     free(transition[i]);
     free(resultat[i]);
    };
    free(transition);
    free(resultat);
    free(echec);
  }
  return 0;
}
```