DAAR PROJET 1

Automate. Clone de egrep avec support partiel des ERE.

Mingtao WANG 21310038

Description

Ce rapport concerne la mise en œuvre des NFA et DFA pour la commande egrep. Egrep prend en entrée une expression régulière et un fichier txt, et renvoie les lignes contenant des segments qui correspondent à l'expression régulière. Certaines conventions de la norme ERE sont assez complexes. Nous nous limiterons aux éléments suivants : parenthèses, choix, concaténation, opération étoile, opération point d'interrogation, opération plus, point (c'est-à-dire caractère universel) et lettres ASCII. Pour cette réalisation, nous avons choisi d'utiliser le langage Python.

Définition du problème

Expression régulière (RegEx)

Les expressions régulières (souvent abrégées en regex ou regexp) sont un outil puissant pour la correspondance de textes, permettant aux développeurs de rechercher, remplacer ou même éditer du texte à l'aide de modèles de séquences de caractères prédéfinies. Une expression régulière est un motif de texte spécifique qui décrit une série de chaînes possibles.

Notre projet d'aujourd'hui est d'étudier les expressions régulières étendues. Pour réaliser notre étude de manière précise et concise, nous limiterons les expressions régulières étendues aux éléments suivants : parenthèses, choix, opération étoile, opération point d'interrogation, opération plus et lettres ASCII. Les explications détaillées sont les suivantes :

- Les parenthèses (parenthèses) : () sont utilisées pour grouper des expressions.
  - Par exemple, a(bc)+ correspondra à "abc", "abcbc", "abcbcbc", etc.
- L'alternative (opération ou) : | est utilisée pour correspondre à n'importe quelle expression des deux côtés de ce symbole.
  - Par exemple, a|b correspondra à "a" ou "b".
- L'opération étoile (opération étoile) : * indique que le caractère ou la sous-expression précédente peut apparaître zéro fois ou plusieurs fois.
  - Par exemple, a* correspondra à "", "a", "aa", "aaa", etc.
- L'opération plus (opération plus) : + indique que le caractère ou la sous-expression précédente peut apparaître une fois ou plusieurs fois.
  - Par exemple, a+ correspondra à "a", "aa", "aaa", etc.
- L'opération point d'interrogation (opération point d'interrogation) : ? indique que le caractère ou la sous-expression précédente peut apparaître zéro fois ou une fois.
  - Par exemple, ab? correspondra à "a" ou "ab".
- La lettre ASCII : représente n'importe quel caractère dans le codage ASCII.
  - Par exemple, a correspondra au caractère "a".

Automates Finis (Finite Automata)

Un automate fini est une autre manière de décrire un ensemble de chaînes de caractères. Il est composé d'un ensemble fini d'états internes et d'un ensemble de règles de contrôle. À partir de l'état actuel et du prochain signal d'entrée, il détermine le prochain état. Les automates finis non déterministes (NFA) et les automates finis déterministes (DFA) sont tous deux des sous-ensembles d'automates finis. Plus tard, nous découvrirons également que le DFA est un sous-ensemble du NFA.

Analyse d'algorithmes

Puisque les expressions régulières et les automates finis peuvent tous deux représenter des ensembles de chaînes, ils devraient pouvoir être convertis l'un en l'autre. En fait, il existe de nombreux algorithmes qui transforment les expressions régulières en automates finis. Je vais maintenant introduire l'une d'entre elles : la construction de Thompson. Cette méthode permet de transformer une expression régulière en un automate fini non déterministe correspondant (en réalité, les expressions régulières peuvent également être transformées en automates finis déterministes correspondants).

Dans le cours DAAR, pour réaliser une recherche d'expressions régulières dans un texte, nous suivons cinq étapes. Je vais vous expliquer chacune de ces étapes et comment les implémenter en code.

RegEx -> RegexTree

La notation post-fixée (ou notation polonaise inversée) est une manière de représenter les expressions arithmétiques sans avoir à se soucier des priorités des opérateurs. Dans cette notation, chaque opérateur suit ses opérandes, contrairement à la notation usuelle où l'opérateur est entre les opérandes.

Prenons l'exemple de l'expression régulière : S(a|g|r)+on. Pour la convertir en notation postfixée :

1. *"S" reste "S".*

2. *a|g|r devient "agr||".*

3. *Le "+" après les parenthèses est représenté par "+", donc "agr||+".*

4. *Enfin, ajoutez "on" à la fin.*

L'expression devient : **Sagr||+on.**

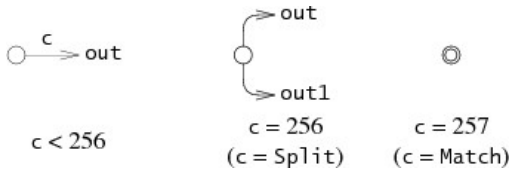
Dans cet exemple, nous voyons comment la notation postfixée simplifie la complexité de l'analyse et de l'évaluation. Il n'y a pas de parenthèses, pas de problèmes de priorité des opérateurs. Il suffit de balayer l'expression de gauche à droite, en utilisant une pile pour traiter les opérandes et les opérateurs.

RegexTree->NDFA

Dans l'article de Thomans, le NFA est défini comme un graphe lié composé de states ayant seulement deux sorties. C'est très similaire à un Arbre. Ainsi, nous abstrayons les états dans le NFA comme suit :

```
class State(object):
        Id  (int)   : représente l'identifiant unique de l'état
        c (char) représente le caractère de l'état. Quand c<256, cela représente le caractère à matcher. Quand c=256, cela représente l'état SPLIT. Quand c=257, cela représente l'état MATCH.
        out (State): représente le prochain état connecté
        out1 (State): utilisé seulement quand c est en état SPLIT. Dans ce cas, out et out1 sont utilisés pour connecter deux choix d'états différents.
        lastlist (int): utilisé pour identifier la liste d'états où se trouve l'état. Pendant l'exécution, cela évite les ajouts répétés.
```

Selon le caractère c dans Etat, on peut diviser Etat en trois types suivants :

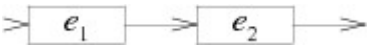


Lorsque le compilateur analyse l'expression postfixée, il maintient une pile de fragments NFA déjà calculés. Les littéraux poussent de nouveaux

fragments NFA sur la pile, tandis que les opérateurs retirent des fragments de la pile puis poussent un nouveau fragment. Par exemple, lors de la compilation de Sagr dans Sagr||+on., la pile contient des fragments NFA pour S, a, g et r. La compilation du | qui suit retire les fragments NFA de g et r de la pile et pousse un fragment NFA représentant la sélection gr. Chaque fragment NFA est défini par son état de départ et ses flèches sortantes :

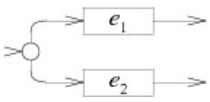
```
class Frag(object):
        debut: représente l'état de départ dans le fragment NFA
        sorties: représente tous les états dans le fragment NFA qui ne sont pas encore connectés. Comme Python passe par valeur, nous passons un tuple (sorties, type_sortie) et utilisons sorties.type_sortie pour se connecter.
        À la fin de la génération du NFA, tous ces états non connectés devraient être connectés à l'état MATCH.
        La classe Fragment est utilisée pour simplifier la génération du NFA. En mettant en cache toutes les sorties du fragment NFA, au lieu de parcourir la liste des états à chaque fois pour obtenir les états non connectés, cela peut grandement accélérer la vitesse d'exécution.
```

Avec les définitions ci-dessus, nous n'avons qu'à suivre les étapes de la construction de Thompson pour construire avec précision le NFA. La fonction post2nfa met en œuvre cette méthode.

a. Pour l'opérateur de concaténation . : 

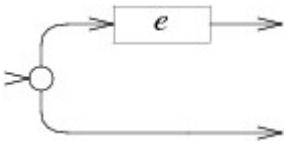

```
if item == '.':
    e2 = frags.pop()
    e1 = frags.pop()
    self.patch(e1.out, e2.start)
frags.append(RegExp.Frag(e1.start, e2.out))
```

b. Pour l'opérateur de choix | :



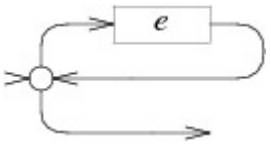

```
if item == '|':
  e2 = frags.pop()
  e1 = frags.pop()
  s = RegExp.State(self.get_state_num(), RegExp.SPLIT)
  s.out = e1.start
  s.out1 = e2.start
  frags.append(RegExp.Frag(s, self.append(e1.out, e2.out)))
```

c. Pour le modificateur de quantité ? :



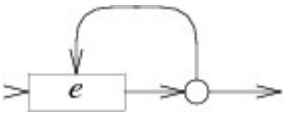

```
if item == ' ?' :
    e = frags.pop()
    s = RegExp.State(self.get_state_num(), RegExp.SPLIT)
    s.out = e.start
    frags.append(RegExp.Frag(s, self.append(e.out, self.singletonlist((s, 'out1')))))
```

d. Pour le modificateur de quantité * :

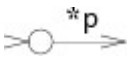



```
if item == '*':
    e = frags.pop()
    s = RegExp.State(self.get_state_num(), RegExp.SPLIT)
    s.out = e.start
    self.patch(e.out, s)
    frags.append(RegExp.Frag(s, self.singletonlist((s, 'out1'))))
```

e. Pour le modificateur de quantité + :




```
if item =='+' :
    e = frags.pop()
    s = RegExp.State(self.get_state_num(), RegExp.SPLIT)
    s.out = e.start
    self.patch(e.out, s)
    frags.append(RegExp.Frag(e.start,self.singletonlist((s, 'out1'))))
```

f. Pour un caractère unique : 


```
s = RegExp.State(self.get_state_num(), item)
frags.append(RegExp.Frag(s, self.singletonlist((s, 'out'))))
```

Après avoir analysé l'expression post-fixée selon ces étapes, tous les pointeurs d'état encore non connectés seront liés à l'état de fin (MATCH), permettant ainsi d'obtenir le modèle NFA correspondant à l'expression régulièr.

Apres analyse l’algorithme, on peux avoir l’avantage l’inconvénients et de cette algorithne :

Avantages :

1. Efficacité : La méthode de construction NFA de Thompson garantit une complexité temporelle linéaire pour la

correspondance des expressions régulières. Cela signifie que pour une entrée de longueur n et une expression régulière de longueur m, la complexité temporelle dans le pire des cas est O(n×m).

2. Simplicité : Cette méthode construit un petit fragment NFA pour chaque élément de l'expression régulière, puis combine ces fragments pour former le NFA complet. Cette approche est à la fois concise et facile à comprendre.

Inconvénients :

1. Indétermination : Un NFA peut avoir plusieurs états actuels, ce qui signifie qu'il peut être nécessaire de suivre plusieurs chemins possibles lors de l'exécution de la correspondance.

Utilisation de l'algorithme de recherche par expression régulière pour correspondre à une chaîne de caractères entrée (optionnel).

Lors de la correspondance avec un NFA, nous avons deux stratégies principales. La première, la méthode de rétroaction, consiste à essayer une branche et, si elle échoue, à revenir en arrière pour essayer une autre, avec une complexité allant jusqu'à O(2^n). La seconde, proposée par Thompson en 1968, teste plusieurs branches simultanément, avec une complexité de O(n), ce qui est plus efficace. C'est cette dernière méthode que nous souhaitons mettre en œuvre.

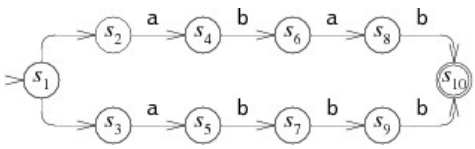
Lors de l'utilisation de NFA pour correspondre à une chaîne de

caractères entrée, nous construisons deux listes : la liste des états correspondants actuels, cstates, et la liste des états correspondants après avoir lu le caractère suivant, nstates. La cstates est initialisée à l'aide du NFA entré. En lisant chaque caractère de la chaîne d'entrée, le système exécute la méthode step. Cette méthode parcourt cstates pour obtenir tous les états qui peuvent correspondre au caractère actuel et ajoute l'état pointé par le pointeur out de chaque état à nstates. Une fois la fonction step terminée, cstates est remplacée par nstates, et nstates est remise à zéro. Enfin, la méthode is_match détermine si, après avoir parcouru toute la chaîne d'entrée, l'état final correspondant de la liste contient un état terminal. Si c'est le cas, la correspondance est considérée comme réussie, sinon, elle échoue.

NDFA->DFA, Minimized DFA et recherche par expression régulière

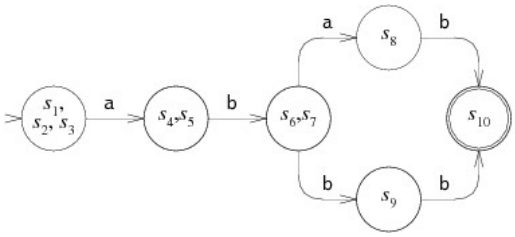
Cependant, le DFA est bien plus efficace que le NFA car, après avoir rencontré une lettre, le DFA n'entrera que dans un état spécifié, contrairement au NFA qui peut entrer dans plusieurs états. De plus, le DFA n'a pas de transitions nulles. Chaque NFA peut être converti en DFA. Nous allons maintenant suivre l'approche de Thomans pour convertir le NFA en DFA.

Pour exemple , le NFA pour (aaba)|(aabb) est :



abab | abbb

Et pour DFA est :



abab | abbb

L'algorithme de recherche fonctionne comme un DFA où cstates est un état DFA. Chaque appel à step produit un nouvel état DFA. L'astuce de Thompson est de cacher nstates après l'exécution de step pour éviter les calculs répétitifs.

D'abord, introduisons un nouveau type de données appelé DState, qui est utilisé pour représenter un état DFA :

```
class DState(object):
    states : Dans le DFA, un état est équivalent à un ensemble d'états (fragments) dans le NFA.
    next : Un dictionnaire dont la clé est la valeur ASCII d'un caractère et la valeur est le DSTATE suivant.
    left : Branche gauche de l'arbre.
    right : Branche droite de l'arbre.          '''
```

Pendant la correspondance, le système vérifie d.next[c] pour chaque caractère d'entrée. Si nécessaire, il génère l'état DFA. La correspondance est réussie si l'état "state" est un état de correspondance. Sinon, elle échoue.

L'arbre de recherche est construit à partir des états NFA triés. En passant cette liste, on obtient le DState associé. Le DFA est donc construit en parallèle avec la correspondance, en se basant sur le NFA.

Ainsi, nous pouvons dire que nous construisons nos données DFA en même temps que nous faisons correspondre la chaîne, en nous basant sur le NFA.

Apres analyse l'algorithme, on peux avoir l'avantage l'inconvénients et de cette algorithne :

Avantages :

1. Déterminisme : Contrairement à NFA, DFA n'a qu'un seul état actuel à tout moment. Cela rend l'exécution de DFA plus simple et rapide.
2. Pas de retour en arrière : En raison de son déterminisme, DFA ne nécessite pas de retour en arrière lors de la correspondance.

Inconvénients :

1. Temps de construction : Bien que le temps d'exécution de DFA puisse être très rapide, le processus de construction peut être relativement lent.
2. Utilisation de la mémoire : En raison du grand nombre possible d'états, DFA peut nécessiter beaucoup de mémoire pour le stockage.

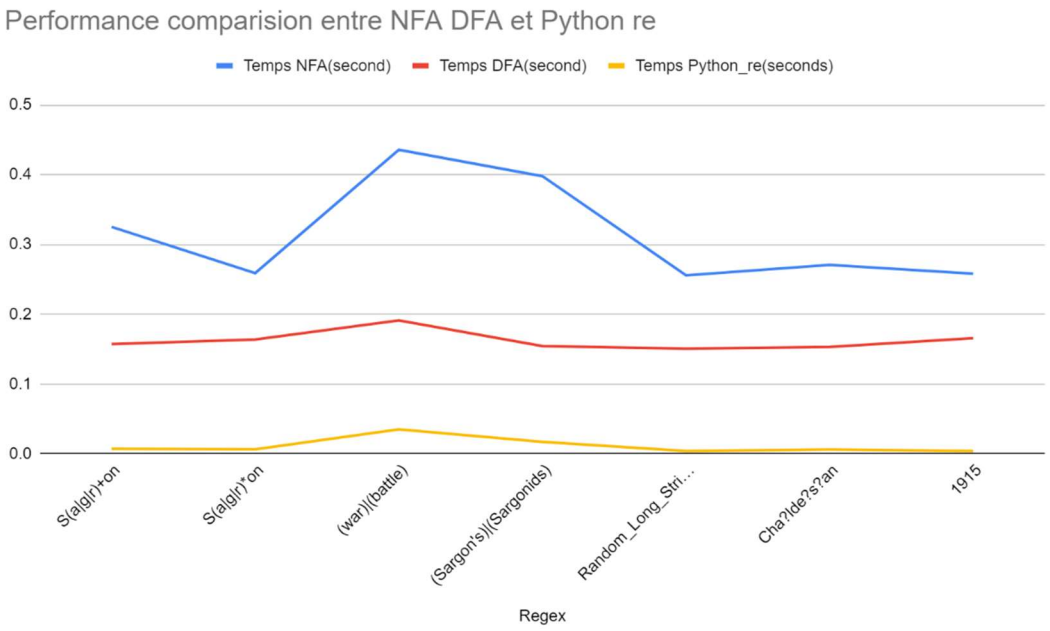
Améliorations :

Afin de réduire la consommation de mémoire, nous avons apporté des modifications sur la base du travail de Thomans. Le "next" est maintenant un dictionnaire, contrairement au tableau de taille 256

mentionné dans l'article de Thomans. Cela permet d'éviter la mémoire inutilisée pour des indices non valides et offre une excellente extensibilité (capable d'accepter des caractères spéciaux avec une valeur ASCII de 257 ou plus).

Essais et analyse des résultats.

Nous avons choisi plusieurs expressions régulières (regex) et le fichier 56667-0.txt comme base de test. Dans les regex testées, nous avons veillé à inclure des lettres, des chiffres, des +, des *, des |, et des parenthèses. Voici les résultats des tests.



Dans les résultats, nous avons constaté que le DFA est plus efficace que le NFA, quel que soit le regex utilisé, en particulier lorsque le regex comprend des opérations de sélection. Lors de la construction du NFA, en présence d'une opération de sélection, il est facile de rencontrer le même caractère mais menant à des états différents. Si on emprunte le

mauvais chemin, cela peut engendrer une perte de temps significative. Bien que dans l'article de Thompson, nous n'ayons pas utilisé la méthode de backtracking, mais plutôt testé plusieurs branches simultanément (ce qui est plus rapide), le DFA demeure plus efficace.

Conclusion

Cette étude a approfondi les principes fondamentaux et la mise en œuvre des expressions régulières et des automates finis. En analysant en détail la méthode de construction de NFA par Thompson et la création de DFA, nous avons pu comprendre en profondeur l'efficacité et la praticabilité des correspondances d'expressions régulières. La méthode de Thompson, en particulier sa stratégie de conversion de NFA en DFA, offre une mise en œuvre efficace, concise et pratique pour les expressions régulières. Cependant, comme toutes les technologies, elle a ses limites, notamment les problèmes d'explosion d'états et de débordement de mémoire qui peuvent survenir lors du traitement d'expressions régulières complexes. Nous avons également apporté quelques améliorations à ce sujet.

Reference :

https://swtch.com/~rsc/regexp/regexp1.html
https://www.codeproject.com/Articles/5412/Writing-own-regular-expression-parser#Seven