Un pequeño tutorial para FAdo

El sistema FAdo es un conjunto de herramientas para la manipulación de lenguajes regulares.

Los lenguajes regulares pueden representarse mediante expresiones regulares (regexp) o autómatas finitos, entre otros formalismos. Los autómatas finitos pueden ser deterministas (DFA) o no deterministas (NFA). En FAdo, estas representaciones se implementan como clases de Python. Aquí se encuentra una documentación completa de todas las clases y métodos.

Para trabajar con FAdo, después de la instalación, importe los siguientes módulos en un intérprete de Python:

[2]:
```
>>> from FAdo.fa import *
>>> from FAdo.reex import *
>>> from FAdo.fio import *
```

El módulo fa implementa las clases para autómatas finitos y el módulo reex las clases para expresiones regulares. El módulo fio implementa métodos para la entrada/salida de autómatas y modelos relacionados.

Convenciones generales

Los métodos cuyo nombre termina en P prueban si el objeto verifica una propiedad dada y devuelven Verdadero o Falso.

Autómatas finitos

La clase superior para autómatas finitos es la clase FA, que tiene dos subclases principales: OFA para autómatas finitos unidireccionales y la clase TFA para autómatas finitos bidireccionales. La clase OFA implementa la estructura básica de un autómata finito compartida por los DFA y los NFA. Esta clase define los siguientes atributos:

`Sigma`: el alfabeto de entrada (conjunto)

`States`: la lista de estados. Es una lista en la que cada estado se referencia por su índice cada vez que se utiliza (transiciones, Final, etc.).

`Initial`: el estado inicial (o un conjunto de estados iniciales para NFA). Es un índice o una lista de índices.

`Final`: el conjunto de estados finales. Es una lista de índices.

En general, no se deben crear instancias (objetos) de la clase OFA. Las clases DFA y NFA implementan DFA y NFA, respectivamente. La clase GFA implementa NFA generalizados que se utilizan en la conversión entre autómatas finitos y expresiones regulares. Las tres clases heredan de la clase OFA.

Para cada clase hay métodos especiales para agregar/eliminar/modificar símbolos del alfabeto, estados y transiciones.

DFA¶

El siguiente ejemplo muestra cómo construir un DFA que acepta las palabras de {0,1}* que son múltiplos de 3.

[3]:
```
>>> m3 = DFA()
>>> m3.setSigma(['0','1'])
>>> m3.addState('s1')
>>> m3.addState('s2')
>>> m3.addState('s3')
>>> m3.setInitial(0)
>>> m3.addFinal(0)
>>> m3.addTransition(0, '0', 0)
>>> m3.addTransition(0, '1', 1)
>>> m3.addTransition(1, '0', 2)
>>> m3.addTransition(1, '1', 0)
>>> m3.addTransition(2, '0', 1)
>>> m3.addTransition(2, '1', 2)
```

Ahora es posible, por ejemplo, ver la estructura del autómata o probar si éste acepta una palabra.

[4]:
```
>>> m3
```

[4]:
```
DFA(['s1', 's2', 's3'], ['0', '1'], 's1', ['s1'], "[('s1', '0', 's1 '), ('s1', '1', 's2'), ('s2', '0
```

[5]:
```
>>> m3.evalWordP("011")
```

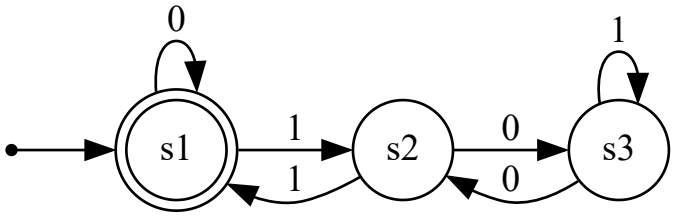
[5]:
```
Verdadero
```

[6]:
```
>>> m3.evalWordP("1011")
```

[6]:
```
FALSO
```

Si está instalado graphviz también es posible visualizar el diagrama de un autómata de la siguiente manera:

[7]:
```
>>> m3.display()
```



En lugar de construir el DFA directamente, podemos cargarlo (y guardarlo) en un formato de texto simple. Para el autómata anterior, la descripción será:

@DFA 0
0 1 1
0 0 0
1 1 0
1 0 2
2 1 2
2 0 1

Entonces, si esta descripción se guarda en el archivo `mul3.fa`, tenemos

[8]:
```
a = "@DFA 0 \n 0 1 1 \n 0 0 0 \n 1 1 0 \n 1 0 2 \n 2 1 2 \n 2 0 1\n"
with open("mul3.fa",'w') as f: f.write(a)
```

[9]:
```
>>> m3 = readFromFile('mul3.fa')
```

Como el conjunto de estados está representado por una lista de Python, el método de lista len se puede utilizar para determinar la cantidad de estados de un FA:

[10]:
```
>>> len(m3.States)
```

[10]: 3

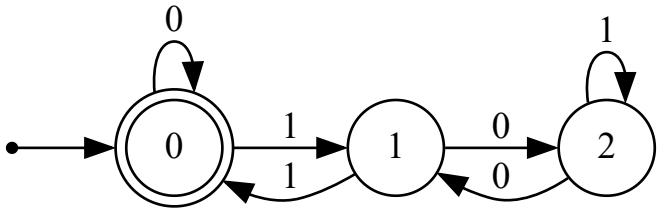
Para el número de transiciones se debe utilizar el método countTransitions()

[11]:
```
>>> m3.countTransitions()
```

[11]:
```
6
```

Los autómatas se pueden visualizar

[12]:
```
>>> m3.display()
```



Para minimizar un DFA se puede utilizar cualquiera de los algoritmos de minimización implementados:

[13]:
```
>>> min = m3.minimalHopcroft()
>>> min
```

[13]:
```
DFA((['0', '1', '2'], ['0', '1'], '0', ['0'], "[('0', '1', '1'), ('0', '0', '0'), ('1', '1', '0'), ('1
```

En este caso, el `DFA m3``.was already minimal so min has the same number of states as

En FAdo se implementan varias operaciones de DFA (que preservan la regularidad): booleana (unión (| o **or**), intersección (& o **and**) y complementación (~ o **invert**)), concatenación (concat), inversión (reversal) y estrella (star).

[14]:
```
>>> u = m3 | ~m3
>>> u
```

[14]:
```
DFA((['0', '5', '10'], ['0', '1'], '0', ['0', '5', '10'], "[('0', '0', '0'), ('0', '1', '5'), ('5', '6
```

[15]:
```
>>> m = u.minimal()
>>> m
```

[15]:
```
DFA((['0'], ['0', '1'], '0', ['0'], "[('0', '0', '0'), ('0', '1', '0')]"))
```

Los nombres de los estados se pueden cambiar en el lugar usando:

```
[16]: >>> m.renameStates(range(len(m)))
[16]: DFA((['0'], ['0', '1'], '0', ['0'], "[(0, '0', 0), (0, '1', 0)]"))
```

Tenga en cuenta que `m`reconoce todas las palabras del alfabeto `{0,1}`. Es posible generar una palabra reconocible por un autómata (`witness`)

```
[17]: >>> u.witness()
[17]: '@épsilon'
```

En este caso esto permite garantizar que usted reconozca la palabra vacía.

Este método también es útil para obtener un testigo de la diferencia de dos DFA (witnessDiff).

Para probar si dos DFA son equivalentes se puede utilizar el operador == (equivalenciaP).

NFA¶

Los NFA se pueden construir y manipular de manera similar. No hay distinción entre NFA con y sin transiciones épsilon. Pero es posible comprobar si un NFA tiene transiciones épsilon y convertir un NFA con transiciones épsilon en un NFA (equivalente) sin ellas.

Conversión entre NFA y DFA

El método toDFA permite convertir un NFA en un DFA equivalente mediante el método de construcción de subconjuntos. El método toNFA migra de manera trivial un DFA a un NFA.

Expresiones regulares

A regular expression can be a symbol of the alphabet, the empty set (@epmtyset), the empty word (@epsilon) or the concatenation or the union (+) or the Kleene star (*) of a regular expression. Examples of regular expressions are a+b, (a+ba)*, and (@epsilon+ a)(ba+ab+@emptyset).

The class regexp is the base class for regular expressions and is used to represent an alphabet symbol. The classes epsilon and emptyset are the subclasses used for the empty set and empty word, respectively. Complex regular expressions are concat, disj, and star.

As for DFAs (and NFAs) we can build directly a regular expressions as a Python class:

[18]:
```
>>> r = CStar(CDisj(CAtom("a"), CConcat(CAtom("b"), CAtom("a"))))
>>> print(r)
```

```
(a + (b a))*
```

But we can convert a string to a regexp class or subclass, using the method str2regexp.

[19]:
```
>>> r = str2regexp("(a+ba)*")
>>> print(r)
```

```
(a + (b a))*
```

For regular expressions there are several measures available: alphabetic size, (parse) tree size, string length, number of epsilons and star height. It is also possible to explicitly associate an alphabet to regular expression (even if some symbols do not appear in it) (`setSigma`)

There are several algebraic properties that can be used to obtain equivalent regular expressions of a smaller size. The method reduced transforms a regular expression into one equivalent without some obvious unnecessary epsilons, emptysets or stars.

Several methods that allows the manipulation of derivatives (or partial derivatives) by a symbol or by a word are implemented. However, the basic class RegExp does not deal with regular expressions module ACI properties (associativity, commutativity and idempotence of the union) , a so it is not possible to obtain all word derivatives of a given regular expression. This is not the case for partial derivatives.

To test if two regular expressions are equivalent the method compare can be used.

[20]:
```
>>> r.compare(str2regexp("(a*(ba)*a*)*"))
```

[20]:
```
True
```

There several methods to convert regular expressions into equivalent nfas or dfas: Thompson, Position/Glushkov, Partial Derivatives (PD), etc.

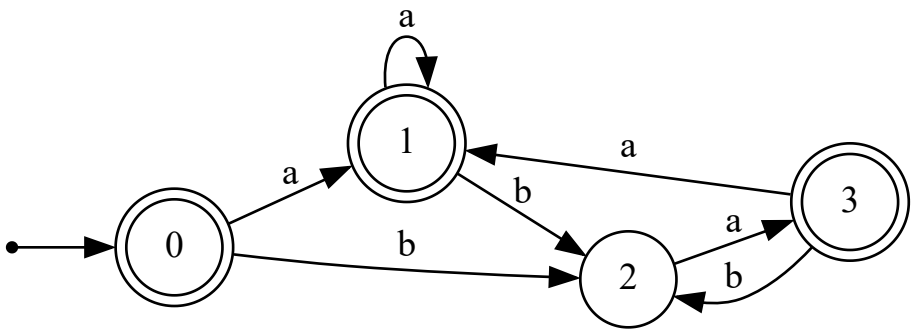
[21]:
```
>>> r.nfaThompson()
```

[21]:
```
NFA(["(0, (0, '0'))", "(0, (0, '1'))", '(0, (1, 0))', '(0, (1, 1))', '(1, 0)', '(1, 1)', '(1, 2)', '(
```

[22]:
```
>>> r.nfaPosition()
```

[22]:
```
NFA((['Initial', "('a', 1)", "('b', 2)", "('a', 3)"], ['a', 'b'], ['Initial'], ['Initial', "('a', 1)",
```

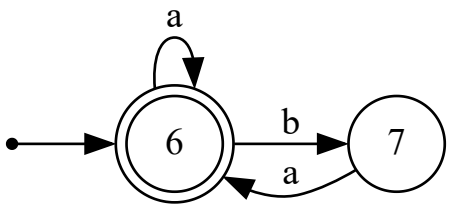
[23]:
```
>>> r.nfaPosition().display()
```



[24]:
```
>>> r.nfaPD()
```

[24]:
```
NFA((['6', '7'], ['a', 'b'], ['6'], ['6'], "[(6, 'a', 6), (6, 'b', 7), (7, 'a', 6)]"))
```

[25]:
```
>>> r.nfaPD().display()
```



Converting Finite Automata to Regular Expressions

Import module conversions

[26]:
```
from FAdo.conversions import *
```

For pedagogical purposes, it is implemented a recursive method that constructs a regular expression equivalent to a given DFA (DFA2regexpDijkstra).

[27]:
```
>>> print(DFA2regexpDijkstra(m3))
```

```
((0 + ((@epsilon + 0) (0* (@epsilon + 0)))) + ((1 + ((@epsilon + 0) (0* 1))) ((1 (0* 1))* (1 + (1 (0*
```

Methods based on state elimination techniques are usually more efficient, and produces much smaller regular expressions. We have implemented several heuristics for the elimination order.

[28]:
```
>>> print(FA2regexpCG(m3))
```

```
((0 + (1 1)) + (((1 0) (1 + (0 0))*) (0 1)))*
```

NFA((['‘,’‘,’‘,’‘, ’0’, ‘1’, ‘2’, ‘3’, ‘8’, ‘9’], [‘a’, ‘b’], [‘8’], [‘9’], “[(‘’, ‘@epsilon’, ‘’), (‘’, ‘@epsilon’, 0), (‘’, ‘@epsilon’, ‘9’), (‘’, ‘a’, ‘’), (‘’, ‘@epsilon’, ‘’), (0, ‘b’, 1), (1, ‘@epsilon’, 2), (2, ‘a’, 3), (3, ‘@epsilon’, ‘’), (‘8’, ‘@epsilon’, ‘’), (‘8’, ‘@epsilon’, ‘9’), (‘9’, ‘@epsilon’, ‘8’)]”))

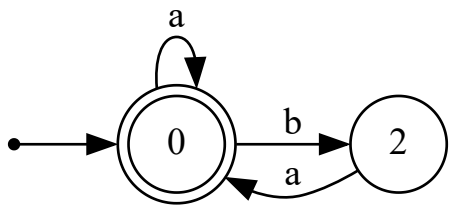
General Example

Considering the several methods described before it is possible to convert between the different equivalent representations of regular languages, as well to perform several regularity preserving operations.

[29]:
```
>>> r.nfaPosition().toDFA().minimal(complete=False)
```

[29]:
```
DFA(['0', '2'], ['a', 'b'], '0', ['0'], "[('0', 'a', '0'), ('0', 'b', '2'), ('2', 'a', '0')]"))
```

[30]:
```
>>> r.nfaPosition().toDFA().minimal(complete=False).display()
```



[31]:
```
>>> m3 == FA2regexpCG(m3).nfaPD().toDFA().minimal()
```

[31]:
```
True
```

More classes and modules

También están disponibles otras clases y módulos, entre ellos:

clase `ICDFArnd`(módulo `rndfa.py`): Generación aleatoria de DFA

clase `FL`(módulo `fl.py`): métodos especiales para lenguajes finitos

Módulo `comboperations.py`: implementación de varios algoritmos para varias operaciones combinadas con DFA y NFA

Módulo `transducers.py`: varias clases y métodos para transductores en formato estándar.

módulo `codes.py`: pruebas de lenguaje para una propiedad (conjunto de lenguajes) especificada por un transductor