Regular Language Library

Grant Clark (November 24, 2022)

Contents

Chapter 1

The STL Helper

1.1 Extended std::to_string Functions

This regular language library relies heavily on extending functionality of the std::to_string functions, and within STL_Helper.h, there are many new std::to_string functions defined. There are defenitions for the following types:

- char
- std::string
- std::pair
- std::unordered_map
- std::unordrered_set
- std::vector

1.2 Extended Hash Structs

This regular language library also relies on extending the structs used to hash items within standard unordrered sets and maps. These hash structs are the reason why it was necessary to extend the usage of the std::to_string functions, as the std::string class already has a defined hash struct. All of my hash structs use the std::to_string functions to hash its values. There are hash structs defined for the following types:

- std::pair

- std::unordered_map
- std::unordered_set

1.3 The "helper" Namespace

The helper namespace was created for the defenition of my std::unordered_set merge function. I used a separate namespace so that the std::merge function can still work, but the version of C++ I was using did not have access to that function, so I put it into its own namespace so that a user can also utilize the std::merge function.

Chapter 2

The DFA Class

2.1 Construction

This class behaves like a deterministic finite-state automata and can compute acceptance for constructed DFAs. To construct a DFA object, you need to provide the constructor with two template types:

- The sigma type, the type that you will use as characters for tranisions, referred to as S_t.
- The state type, the type you will use to represent states, referred to as Q_t.

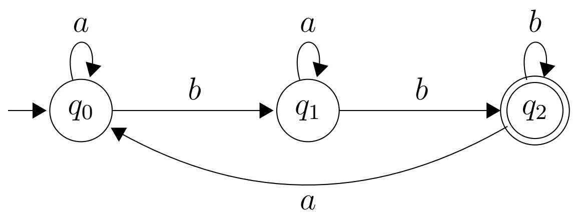
Note: Both given types **MUST** have an std::to_string function overwritten to work, so if you want to use a class that does not already have an std::to_string function written for it, create one for your classes in the STL_Helper.h file.

Next, the constructor must be provided with the following in this order:

- std::unordered_set< S_t >, which is your sigma (set of valid transitioning characters)
- std::unordered_set< Q_t >, which is the set of your states
- Q_t, your initial state
- std::unordered_set< Q_t >, which is the set of accepting states (subset of the set of states)
- std::unordered_map< std::pair< Q_t, S_t >, Q_t > which will act as your delta, or transition function.

This looks rather confusing, so to clarify we will use an example. We will take the following DFA:

$$\Sigma = a, b$$



and represent it using the DFA class, using std::string as both the sigma and state type:

```
#include <iostream>
#include "RegLang.h"

typedef std::string S_t;
typedef std::string Q_t;

int main()
{
    S_t a = "a";
    S_t b = "b";
    Q_t q0 = "q0";
    Q_t q1 = "q1";
    Q_t q2 = "q2";

    std::unordered_set< S_t > S {a, b};
    std::unordered_set< Q_t > Q {q0, q1, q2};
    std::unordered_set< Q_t > F {q2};

    std::unordered_map< std::pair< Q_t, S_t >, Q_t > delta;
    delta[{q0, a}] = q0;
    delta[{q0, b}] = q1;
    delta[{q1, a}] = q1;
    delta[{q1, b}] = q2;
    delta[{q2, a}] = q0;
    delta[{q2, b}] = q2;

    DFA< S_t, Q_t > M(S, Q, q0, F, delta);
}
```

You can also construct a DFA using the copy constructor like so:

```
DFA< S_t, Q_t > M(S, Q, q0, F, delta);
DFA< S_t, Q_t > M_2(M); //M_2 is now a copy of DFA M.
```

You can also use the assignment operator to make a DFA into a copy of another DFA:

```
DFA< S_t, Q_t > M(S, Q, q0, F, delta);

// Assuming S_, Q_, q0_, F_, and delta_ are all valid.
DFA< S_t, Q_t > M_2(S_, Q_, q0_, F_, delta_);

M_2 = M; //M_2 is now a cppy of DFA M.
```

2.2 Accessing Members

The DFA object has 5 methods used to access members, however, once a DFA is created, you are not allowed to alter the contents of its members. i.e., once a DFA is made, it meant is immutable (the exception to this is the assignment operator). Each of the following methods returns that given member variable as a constant reference:

- sigma()
- states()
- initial_state()
- accepting_states()
- delta()

2.3 Computing Acceptance

A DFA object can be used to compute acceptance by providing the operator() method with a string of characters. This string is represented represented by an **std::vector**< **S_t** >, and currently is the only accepted input for acceptance. Using the example DFA we created above, we can compute the following:

```
// S, Q, q0, F, and delta are all the same
// from the previous example.
DFA< S_t, Q_t > M(S, Q, q0, F, delta);

M({a, b, a, a, b}); // True
M({b, b, a, b});    // False
M({a, a, a, a, a}); // False
M({b, b, a, b, b}); // True
M({a, b, b, b, b}); // True
```

Errors associated with operator():

- **DFA_Invalid_Sigma_Character_Error**, This error is thrown if you give the DFA an std::vector of characters to compute that contains a character that is not within its given set of characters Sigma.

2.4 Instantaneous Descriptions of Acceptance

You can view the process of acceptance visually using the **IDs()** method. This function takes in an std::vector of characters in sigma to compute acceptance, but instead of returning a boolean, it returns a, std::vector< std::string > which contains an instantaneous description of each step in the computation. Using the given example DFA above, this code:

```
DFA< S_t, Q_t > M(S, Q, q0, F, delta);

std::vector< S_t > w {a, b, a, a, b};

std::vector< std::string > ids = M.IDs(w);
for (const std::string & str : ids)
    std::cout << str << std::endl;
```

will produce this output:

```
[a,b,a,a,b]
(q0, [a,b,a,a,b])
(q0, [b,a,a,b])
(q1, [a,a,b])
(q1, [a,b])
(q1, [b])
(q2, [])*
```

The first string in the vector of strings returned by IDs() is always just the string itself, and every subsequent line is a computational step, with the first computational step always being a tuple of the initial state and the initial string given. Each subsequent step uses the first character in the string and moves to the next state using delta until there are no characters left in the string. If the final state you have reached is an accepting state, the final tuple will have a * character printed after it to denote that the string was accepted.

NOTE: All errors associated with operator() and how they are thrown are the same for the IDs() function.

2.5 Validating a DFA

This class includes a method, **valid()**, which returns a boolean that denotes whether the given DFA is a valid DFA. For the DFA to be considered valid, it must meet the following criteria:

- Set of states must be non-empty.
- All accepting states must be present within the set of states.
- Initial state must be within the set of states.
- All states and characters within delta transitions must be within the set of states and set of characters sigma respectively.
- All states transitioned to within delta must be within the set of states.
- Each state must have exactly one transition for each character in the set of characters sigma.
- Each state must be reachable.

If all of the above criteria are met, the **valid()** function will return true.

2.6 DFA Compliment

The **compliment()** method returns a DFA< S_t, Q_t > object in which every accept state of the initial DFA is now no longer an accept state, and every state in the initial DFA which was not an accept state is now an accept state. Using the acceptance examples, we can do the following:

```
// S, Q, q0, F, and delta are all the same
// from the previous example.
DFA< S_t, Q_t > M(S, Q, q0, F, delta);

M({a, b, a, a, b}); // True
M({b, b, a, b});    // False
M({a, a, a, a, a}); // False
M({b, b, a, b, b}); // True
M({a, b, b, b, b}); // True

DFA< S_t, Q_t > M_comp(M.compliment());

// Acceptances are now flipped.
M_comp({a, b, a, a, b}); // False
M_comp({b, b, a, b});    // True
M_comp({a, a, a, a, a}); // True
M_comp({b, b, a, b, b}); // False
M_comp({a, b, b, b, b}); // False
```

2.7 DFA Minimization

the **minimal()** method returns a minimized DFA< S_t, Q_t > using Hopcroft's algorithm for DFA minimization.

```
// S, Q, q0, F, and delta are all the same
// from the previous example.
DFA< S_t, Q_t > M(S, Q, q0, F, delta);

M = M.minimal(); // M has now been minimized.
```

2.8 is_accepting() Method

This method takes in a state object (Q_t) and returns true if it is within the set of accepting states.

2.9 DFA to NFA Conversion

the **to_nfa()** method returns an NFA< S_t, Q_t >. This method requires that the user provide an S_t value to be used as the epsilon character in the character set sigma of the returned NFA.

```
// assuming all given parameters are valid.
DFA< S_t, Q_t > M(S, Q, q0, F, delta);

//In this specific instance, we are assuming that
//S_t is std::string.
S_t epsilon = "";

NFA< S_t, Q_t > N(M.to_nfa(epsilon));
```

2.10 DFA Intersection

This library comes with a function that will take two DFA objects and return a new DFA created from the intersection of the given DFAs. The returned DFA will be in this form: DFA< S_t, std::pair< Q_t0, Q_t1 > > where Q_t0 is the state type of the first DFA, and Q_t1 is the state type of the second DFA. This function takes three templated parameters, the first being the sigma type of both DFA, the second being the state type of the first DFA, and the third being the sigma type of the second DFA. The two parameters it takes in are the two DFA in which the intersection will be performed on.

NOTE: This function only works on two DFA who share the same S_t type. Also, do **NOT** use this function on two DFA who share any state names, as it will result in unintended behavior. Additionally, the two DFA sent in **MUST** have exactly the same sigma, or set of accepted characters, otherwise an intersection does not make sense to perform.

```
// assuming all given parameters are valid and the sets
// of states Q0 and Q1 do not share any state names.
// NOTE: Sigma is shared.
DFA< S_t, Q_t0 > M0(S, Q0, q00, F0, delta0);
DFA< S_t, Q_t1 > M1(S, Q1, q10, F1, delta1);

DFA< S_t, std::pair< Q_t0, Q_t1 > >
    intersection = dfa_intersection< S_t, Q_t0, Q_t1 >(M0, M1);
```

Errors associated with dfa_intersection():

- **DFA_Sigma_Mismatch_Intersection_Error**, This error is thrown if you call the intersection function on two DFA who do not share the same exact set of characters sigma.

Chapter 3

The NFA Class

3.1 Construction

todo