The Flow Tutorial

Flow is a programming language that is designed to aid in specifying graphs and determining their properties. In this tutorial, we will guide you through writing a graph that represents a DFA that recognizes the language consisting of the single string "hello world!" from the English alphabet. We will also be writing a solver that uses this graph to compute if an arbitrary string from the English alphabet equals "hello world!".

Getting Started

Before we dive right in, we need to first establish the proper mindset for thinking in Flow. A Flow graph is composed of two fundamental components: Nodes and Arcs. Nodes are terminal points on the graph which simulate vertices, and Arcs are connections that link the Nodes together. Both of these fundamental components can have zero or more primitive attributes. One of Flow's defining characteristics is that the graph section and the solver section are completely separate. That is, there is a part of Flow dedicated solely to defining a graph, and another part focused only on solving a graph. So what does that mean for you? When you are writing a graph definition, only concern yourself with representing that graph in the most convenient fashion - do not worry about how you are going to solve it!

As long as certain preconditions are met, which are mostly described by a type definition (to be discussed shortly), the solver should be able to accept any number of graphs and operate on them. For instance, if you write a traveling salesperson solver to do your graph theory homework for you, you can apply it to each graph in your problem set without having to change anything! In the same way, a single graph can be representative of many problems, and can be used in conjunction with multiple solvers. This plug-and-play functionality is what makes Flow so flexible. For example, say you have a large, undirected graph. You can pair it with one solver to find a Euler path, and then pair it with another solver to find a Hamiltonian path without making any changes to the graph itself. You just need to make sure that all three components (graph, Euler solver, and Hamiltonian solver) use the same type definition. Encapsulation, a key engineering feature that ensures scalability, is achieved through clearly delineating a problem's preconditions, initial state, and algorithm, and is the key to Flow's expressiveness and versatility.

One Last Note: Flow Comments

Throughout this tutorial, we will be using comments, which are parts within the code of a program that are ignored by the compiler. Text within Flow code can be 'commented out' by encasing text between the two character symbols '/*' and '*/'. Multiple lines can be commented out in this manner. For example:

```
/* This line is a comment in Flow, and will be disregarded by
the compiler. */
```

An Overview

Now that we are thinking like true Flow programmers, let us take a look at how Flow works. We have three different file types that each need to be used in a very specific way. First is the previously mentioned but unexplained type definition. The type definition, or typedef for short, is a file that describes the preconditions for the problem that you want to solve. In it, you tell Flow what it should expect your graph's nodes and arcs to look like. More specifically, you define what attributes your nodes and arcs have. With this, a programmer (even multiple programmers) can now write the other two files, a graph definition and solver, both self explanatory, that adhere to the specifications in the typedef, and be assured thatthey will be compatible.

In order to specify which typedef the graph definition and solver are following, each file must begin with a line declaring the file name of the typedef. So if a graph definition and solver both "use" the same typedef, then each file is operating under the assumption that the other is properly conforming to the standards set forth in the typedef, and will thus be compatible. Without the ability to assume this, you would not be able to so distinctly separate the graph definition and solver! A programmer working on the solver would need to read through the graph definition in order to understand what they have to work with. It is only through strict adherence to the preconditions set forth in the typedef, and the assumption that all related files do indeed adhere to them, that enables Flow programs to work.

The Typedef

Now let us examine the typedef more closely. There are many different types of graphs. Some graphs have nodes that are associated with numerical values, while others have nodes that are associated with colors, depending on the problem that the graph is representing. The arcs connecting these nodes can also have different attributes depending on the context of the problem. For this reason, it is necessary to tell the compiler about the specific properties of the nodes and arcs you will be dealing with in your Flow program. We will tell the compiler about the specifics of our graph by writing a typedef in a file with extension ".flowt".

In this tutorial, our typedef will need to define the special properties of a deterministic finite automata (DFA). There are 3 properties of a DFA that we must address:

1. Each node must either be marked as an accept state or a non-accepting state.

2. One of the nodes is the special start state.
3. Each arc must be associated with a letter of the English alphabet. (This represents a transition rule.)

Let's do this in a file called "dfa.flowt".

```
/* The typedef section */

/* Each DFA node has two attributes, and each path arc has one
attribute. */

NODE state(string value, int isAccepting) START; //START is a
required node label

ARC path(string symbol);
```

Let's analyze this part by part.

```
NODE state(string value, int isAccepting) START;
```

This line of code specifies that our nodes will be called 'state's. A state will have the attributes 'value', which will be an arbitrary string used to name the node, as well as an int variable 'isAccepting', which tells us if the state is an accepting node or not. Since Flow does not contain the boolean type, each data type in Flow has an inherent truth value. For integers, all values except 0 are true. So, we will use a 1 to represent true, that it is an accepting state, and 0 to represent false, that it is not an accepting state. At the end of the line, the word START defines a label named START. We will have a discussion about labels later.

```
 ARC path(string symbol);
```

Our arcs will be called 'path'. They will have the property 'symbol', which will be a symbol in our language so that the path can represent a transition rule.

The Graph Definition

Now we are ready to write a graph definition that represents a DFA. Our DFA will recognize the language consisting of the single string “hello world!”. Graph definition files have the extension ".flowg", so let us start a new file called "dfa.flowg". We first present the complete code, and then we will break it down and analyze it part by part.

```
use 'DFA.flowt';

/* Building the graph */

START: @s0 "s0", 0;  //state s0 is labeled as the START state
@s1 "s1", 0;
s0 -> s1 "h";
@s2 "s2", 0;
s1 -> s2 "e";
@s3 "s3", 0;
s2 -> s3 "l";
@s4 "s4", 0;
s3 -> s4 "l";
@s5 "s5", 0;
s4 -> s5 "o";
@s6 "s6", 0;
s5 -> s6 " ";
@s7 "s7", 0;
s6 -> s7 "w";
@s8 "s8", 0;
s7 -> s8 "o";
@s9 "s9", 0;
s8 -> s9 "r";
@s10 "s10", 0;
s9 -> s10 "l";
@s11 "s11", 0;
s10 -> s11 "d";
@s12 "s12", 1;
s11 -> s12 "!";
@skill "kill", 0;

List of string lst = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P','Q', 'R', 'S', 'T', 'U',
'V', 'W', 'X', 'Y', 'Z','a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',' ','!'];
```

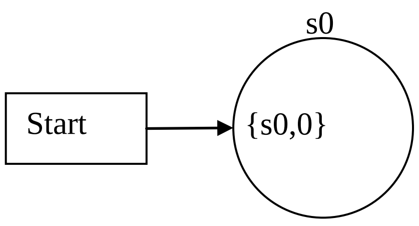
```
List of Node dfaLst =
[s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,skill];
List of string strLst = ["h","e","l","l","o","
","w","o","r","l","d","!",""];

int pos = 0;
while (pos < dfaLst.length){
     int k = 0;
     while (k < lst.length)
     {
          string character = lst[k];
          if(character != strLst[pos])
          {
               n -> skill character;
          }
     }
     pos = pos + 1;
}
```

The first thing we do is tell the graph about the typedef we have written.

```
use 'DFA.flowt';
```

Next, we make a node. Recall that in our typedef, we said '`state(string value, int isAccepting)`'. We can set the node's properties in the order that they are defined in our typedef. You begin a node definition with the '@' character, followed by its name. We set an arbitrary name for this node's identifier, '@s0', and then we will have this correspond with its string value. Since it is not an accepting state, we will also set `int isAccepting` to 0 to represent false. A diagram is included below the code to help you visualize this.

```
START: @s0 "s0", 0;  //state s0 is labeled as the START state
```



Because we added START, we are labeling this node as the start node for our graph.

*****

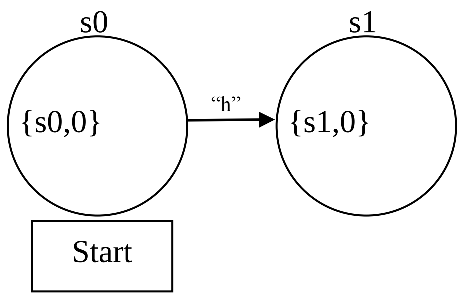
At this point, we must make the distinction between a variable name and a label. A variable name identifies an object or primitive *only within* a graph or solver file, while a label is a special name, created in the typedef, for use *across both* a graph and solver file. Any labeled object in the graph definition file will be accessible by the solver using the same label. Contrast the following two lines.

```
@anode “somestring”, 0; //anode will NOT be visible to the
solver

START: @anode “somestring”, 0; //anode will be visible to the
solver using the label START
```

*****

Returning to our sample program, we create another non-accepting state node, and connect the start node to it with an arc. Since we are trying to recognize “hello world!”, this first arc will represent the transition made upon seeing an 'h' as input. We must therefore set the appropriate property in the arc to 'h'. Again, a diagram follows the code for visualization.

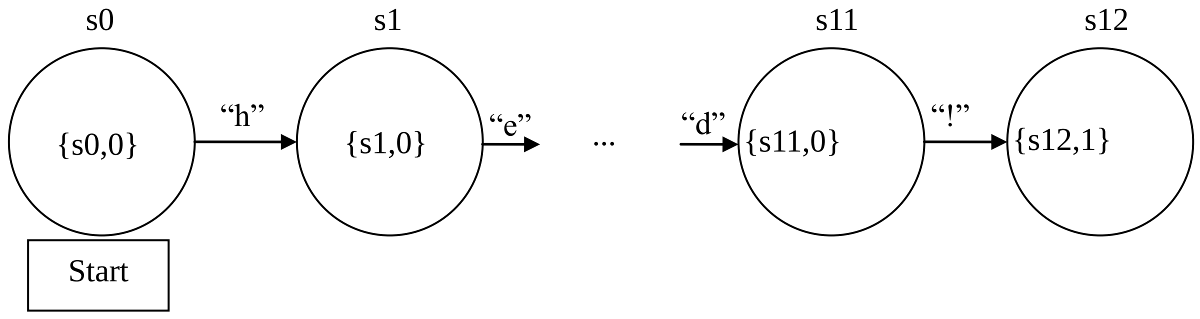
```
@s1 "s1", 0;
s0 -> s1 "h";
```



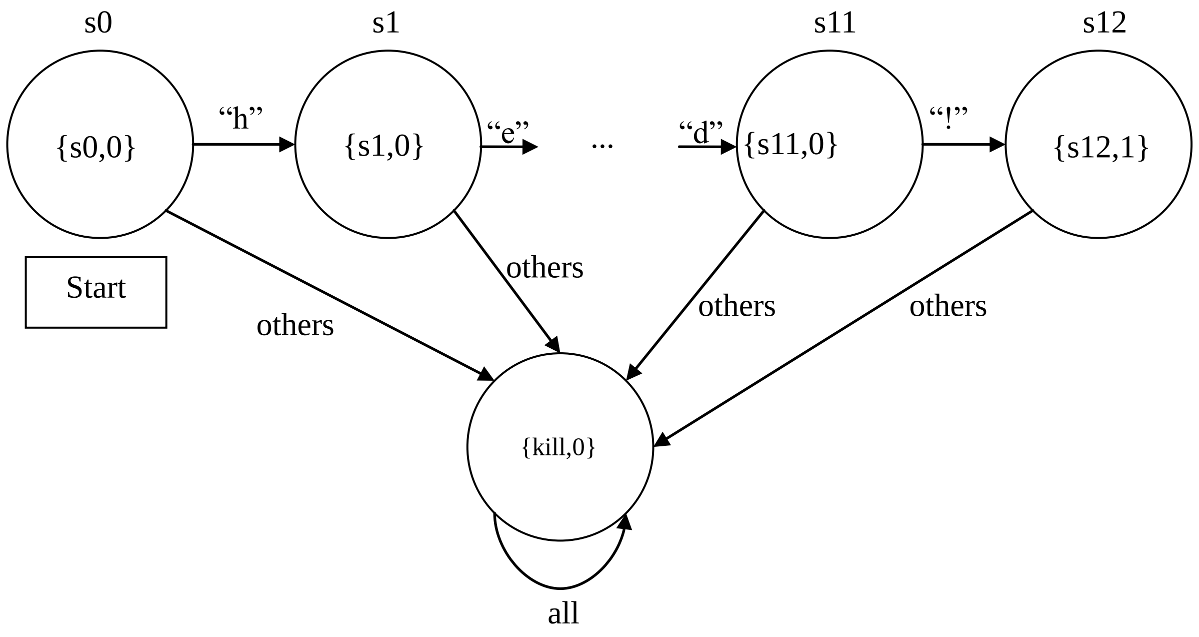
As you can see from the code, an arc is declared by first stating the from-node, then the "->" symbol, and then the to-node. Any subsequent parameters are matched with the attributes declared in the typedef: `ARC path(string symbol);`

We eventually form one long chain, linking each of the letters in the string “hello world!”. The last node in the chain is reached when the input '!' is seen (given that the characters h, e, l, l, o, [space], w, o, r, l, and d were previously seen, in that order). This last node is an accepting-state, so we must set the appropriate property.

```
@s12 "s12", 1;
s11 -> s12 "!";
```



Because each node of the DFA must have a transition for every possible letter of the language, we have to create a "sink" node. The purpose of the sink node is to collect all inputs on which we should not advance along the chain of the DFA. In order to do this, we must visit each node in our DFA. We look at the input which will advance the DFA along the chain, and for all other letters that are not equal to this input, we create a corresponding arc to the sink node. The diagram below shows you what our final DFA graph should look like.



It would be quite a hassle to write each of these connections one by one. To automate the process, we will utilize lists and loops. A Flow list is an array. Each Flow list can hold exactly one type of data. For example, you might have a list to hold strings, and another list to hold Nodes. Lists are declared with the keyword "List of" followed by the type and then its name. Values are defined with square brackets '[' and ']' and separated by commas.

```
List of string ltrLst = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P','Q', 'R', 'S', 'T', 'U',
'V', 'W', 'X', 'Y', 'Z','a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',' ','!'];

List of Node dfaLst = [s0, s1, s2, s3, s4, s5, s6, s7, s8, s9,
s10, s11, skill];

List of string strLst = ["h", "e", "l", "l", "o", " ", "w", "o",
"r", "l", "d", "!", ""];
```

Here we have three lists. `ltrLst` contains the symbols of our alphabet, `dfaLst` contains the path that will be traversed on the correct input, and `strLst` contains the string that will send us on the correct path.

```
int pos = 0;
while (pos < dfaLst.length){
     inner loop omitted
}
```

Here, we have created a loop to traverse all the nodes. This is a standard while loop that you might have seen before in Java or C. For every node that we touch, we connect it to the sink node on all inputs that are not considered 'correct'. Look at the inner loop.

```
     int k = 0;
     while (k < ltrLst.length)
     {
          string character = ltrLst[k];
          if(character != strLst[pos])
          {
               n -> skill character;
          }
     }
     pos = pos + 1;
```

For example, on the first iteration of the loop, the node s0 is visited. Within the inner loop, an arc is created to the sink node on every letter that is not `strLst[0]` (in this iteration, that would be the symbol 'h').

Also note that on the last iteration of the loop, we are visiting the node 'skill'. This is the sink node, located in our list at `dfaLst[12]`. The corresponding symbol in `strLst[12]` is the empty string. Since it will not match any of the symbols in our language (located in `ltrLst`), the sink node is connected to itself on every input.

The Solver: Step-by-Step

Now that we have a graph, we will need some way to solve the graph. We will write a solver in a file with extension ".flow". The first thing to do is to figure out exactly what we want our solver to do. Well, we are going to need a function. Let us call the function “simulate,” since it is going to simulate a DFA. A DFA needs an input from our alphabet (in this case our alphabet is, as mentioned before, both upper and lowercase letters, the space character, and the exclamation mark), so the input should be in the form of a string. The DFA will either accept or not accept, so we can return either true or false, respectively. If it accepts, let us say that it accepts, and if it doesn't accept, let us say that it doesn't accept.

Here is what the code looks like now:

```
string input = "Hello World";
print "Input is '" + input + "'\n";

int truth = 0;
if (simulate(input)) {
     truth = 1;
     print "Input is accepted.\n";
}
if (truth == 0){
     print "Input is not accepted.\n";
}
```

Looking good! However, we still need to build the function itself, which will be the tricky part.

So the first step is to access our start node, in order to start the DFA in the correct position. Luckily, we attached the label START to our first node. In order to assign it to a more useful identifier, simply declare the identifier and assign away. Labels are stored in the structure "Graph" and accessed through the dot accessor ('.').

```
function simulate(string input) {
    @current = Graph.START;
```

After that, we'll need to know about its edges, so that we can go to the correct next node. Luckily, we have the method `current.arcsOut()` to take care of that.

We'll show you what we're going to do with all of those edges, but for now let's just say we have a single edge, `myEdge`. In order to get where the edge is going to, call `myEdge.to`. Similarly, to figure out which node an edge is from, use `myEdge.from`. Here's an example usage:

```
nextNode = myEdge.to;
```

We can now do the same actions on `nextNode` that we could do on `current`. In this case, we're going to print its value.

```
print nextNode.value;
```

Now that we know how to get from node to node, let's start writing the actual function. First we need to say we're in a function, what the parameters are, and get the starting node, which we've seen before.

```
function simulate(string input) {

    @current = Graph.START;
    print "The first current variable is set.";
}
```

This is a little boring. We want to actually go through all of the letters in the input, and go from node to node. In order to do that, we need to iterate over the letters. We'll use a traditional while loop to do that. The while loop in Flow is structured much like many other while loops. In this case, we want our identifier to iterate over the all letters in the string, so we'll use the built-in attribute of strings, their length.

```
function simulate(string input){
     dfa current = START;
     print "The current version is set."

    int i = 0;
    while(i < input.length) {
          //inner loop omitted
          i = i + 1;
     }
}
```

The other kind of control flow we use are if statements. If you want a condition to be met before code is executed, use an if statement. Again, these function much like the if statements found in many other languages.

Let's finish making the DFA solver. For each letter, what we want to do is look through the adjacencies list for the edge with the symbol of the letter that we are currently on. We are guaranteed that it will be there, because for each Node on a DFA, for every character in the alphabet, the behavior of the DFA is well defined, and each symbol will appear exactly once. First we'll declare a next path—after that, we'll look through the adjacent edges, and if one of them has a symbol of the input that we're currently on, we'll set next to be connected. We will only need to do it once. All of our symbols are only one character long, so it is sufficient to choose `input.substring(i,i+1)`.

```
Arc next;
List of Arc pathLst = current.arcsOut();

int j = 0;
while(j < pathLst.length)
{
       Arc connected = pathLst[j];
       if(connected.symbol == input.substring(i, i+1)
       {
          next = connected;
          print "Went to the next edge.";
       }
       j++;
}
```

Next, if the path has been set, then we want to change our `current` to the new `current`, that is, the `current` at the end of the path, `next`.

```
int isNext = 0;
if (next) {
    isNext = 1;
    print "Navigating to the node at the end of that edge.";
    current = next.to;
}
```

If it hasn't been, then we want to exit gracefully, returning that it did not succeed.

```
if (isNext == 0) {
    print "There was a problem.  Now exiting.";
    return 0;
}
```

And that's all there is to the solver!

The Solver: Putting It All Together

Altogether, the code for the solver should look like this:

```
/* This function simulates the dfa specified above on its
parameter input string */

//precondition: dfa, e.g. every character in the alphabet
//is represented uniquely by an out-arc on each node
use 'DFA.flowt';

function simulate(string input) {
    @current = Graph.START;
    print "The first current variable is set.";

    print "Entering the for loop to look through the arcs.";

    int i = 0;
    while(i < input.length) {
       Arc next;
       List of Arc pathLst = current.arcsOut();

       int j = 0;
       while(j < pathLst.length)
       {
          Arc connected = pathLst[j];
          if(connected.symbol == input.substring(i, i+1)
          {
               next = connected;
               print "Went to the next edge.";
          }
          j++;
       }

       int isNext = 0;
       if (next) {
          isNext = 1;
          print "Navigating to node at the end of that edge.";
          current = next.to;
       }
       if (isNext == 0) {
          print "There was a problem.  Now exiting.";
          return 0;
       }
       i = i + 1;
    }

    if (current.isAccepting) {
       return 1;
    }

    return 0;
}

/* All code not within a function definition will execute every
time */

string input = "Hello World";
print "Input is '" + input + "'\n";

int truth = 0;
if (simulate(input)) {
     truth = 1;
     print "Input is accepted.\n";
}
if (truth == 0) {
     print "Input is not accepted.\n";
}
```

Ready, Set, Go!

You have just written your first complete Flow program! Let us go over what we have accomplished. We started with a problem we wanted to solve using Flow: how to simulate a DFA that recognizes the language consisting only of the string "hello world!" To start, we created a typedef to specify the attributes of the nodes, which represented the DFA's states, and the arcs, which represented the DFA's transitions. With this, we were able to then write a graph definition to represent our DFA, and a solver to run simulations over it. Now let us run it and see it in action!

In order to do so, we have to compile the various parts of our program in 3 steps.

1. Compile the graph definition, which results in a Graph.java file.

   ```
   Flow ExampleGraph.flowg
   --> Graph.java
   ```

2. Compile the solver with the newly-created Graph.java file.

   ```
   Flow ExampleSolver.flow Graph.java
   --> Solver.java
   ```

3. Run the solver through Java!

   ```
   java Solver.java
   ```

Now you have a running program! Try entering in a variety of strings, and see what happens. The only time the program should accept your input is when it is the string "hello world!" Give it a try! Congratulations - you have just written and run your first program in Flow!