The Flow Language: An Overview

David Hughes, Mark Liu, William Liu, Moses Nakamura, Stephanie Ng

Problem Domain

The Flow language consists of two aspects, the specification of a graph and the computation of the properties of a graph. The two parts are coded independently of each other, and then compiled together into a program that first creates the graph and then computes the desired properties.

Sample Uses of the Language

1. Min/Max flow calculation
2. Graph traversal algorithms
3. Dijkstra's algorithm
4. DFA simulation

Properties

Easy to use and intuitive: Just define the nodes and arcs of a graph, then describe the information you want to pull out of it.

Versatile: Because the graph definition and the solver are kept in separate files, it is easy to apply a new solver to an existing graph to get additional information out of it. It is also easy to use a tried-and-true solver on a brand new graph.

Useful: If the solution to an algorithm can be coded, then it is possible to write it as a solver in Flow. Solvers for Dijkstra's alrgorithm, Kruskal's algorithm, and depth-first search are easily implemented.

Example Programs

Term Definition

- **Node:** the data type which represents the nodes of the graphs
- **Arc**: the connections between nodes, commonly known as edges in graph theory
- **Label**: a name of a node that persists between the graph definition and solver

File Types

- **Typedef (.flowt) :** fundamental interface between the graph maker and the Solver. *The typedef represents the context of the graph problem. For example, nodes and arcs used in Dijkstra's algorithm have different attributes than nodes and arcs used in max flow problems. These context-dependent attributes must be specified by the typedef.*
- **Graph Definition (.flowg):** the specification for creating a graph

- **Solver (.flow):** a functional specification of the computation to be done on the graph

Example Program #1

/* Typedef */

```
//Define a depthN type node to have 0 attributes, and require 1 of them to be labeled root
NODE depthN( ) root;
//Define depArc type arc to have 0 attributes
ARC depArc( );
```

/* Graph Definition */

```
//Specify the typedef
use 'dfa.flowt';

//Create a depthN type node called 'a', label it with the special label root
ROOT: @a;

//Create two more depthN nodes
@b;
@c;

//Connect node 'a' to 'b', then node 'a' to 'c' with anonymous arcs
a -> b;
a -> c;
```

/* Solver */

```
//Specify the typedef
use 'dfa.flowt';

function depth(Node myNode)
{
        int depth = 0;
        List of Arc arcs = myNode.arcsfrom();
        if(arcs.length == 0) {
                return 0;
        }
        int k = 0;
        while(k < arcs.length) {
                int temp = depth(arcs[k].to);
                if(temp > depth) {
                        depth = temp;
                }
        }
        return depth + 1;
}
print depth(root);
```

Example Program #2

```
/* This graph simulates a DFA that represents all strings of a's and b's with an even number of a's
and an even number of b's. */
```

/* Typedef */

```
//Each state 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);
```

/* Graph Definition */

```
//Specify the typedef
use 'dfa.flowt';

//Building the graph
START: @s0 "s0", 1;  //state s0 is labeled as the START state
@s1 "s1", 0;
@s2 "s2", 0;
@s3 "s3", 0;

//Path definition
s0 -> s1 "a";
s1 -> s0 "a";
s1-> s2 "b";
s2 -> s1 "b";
s2 -> s3 "a";
s3 -> s2 "a";
s3 -> s0 "b";
s0 -> s3 "b";
```

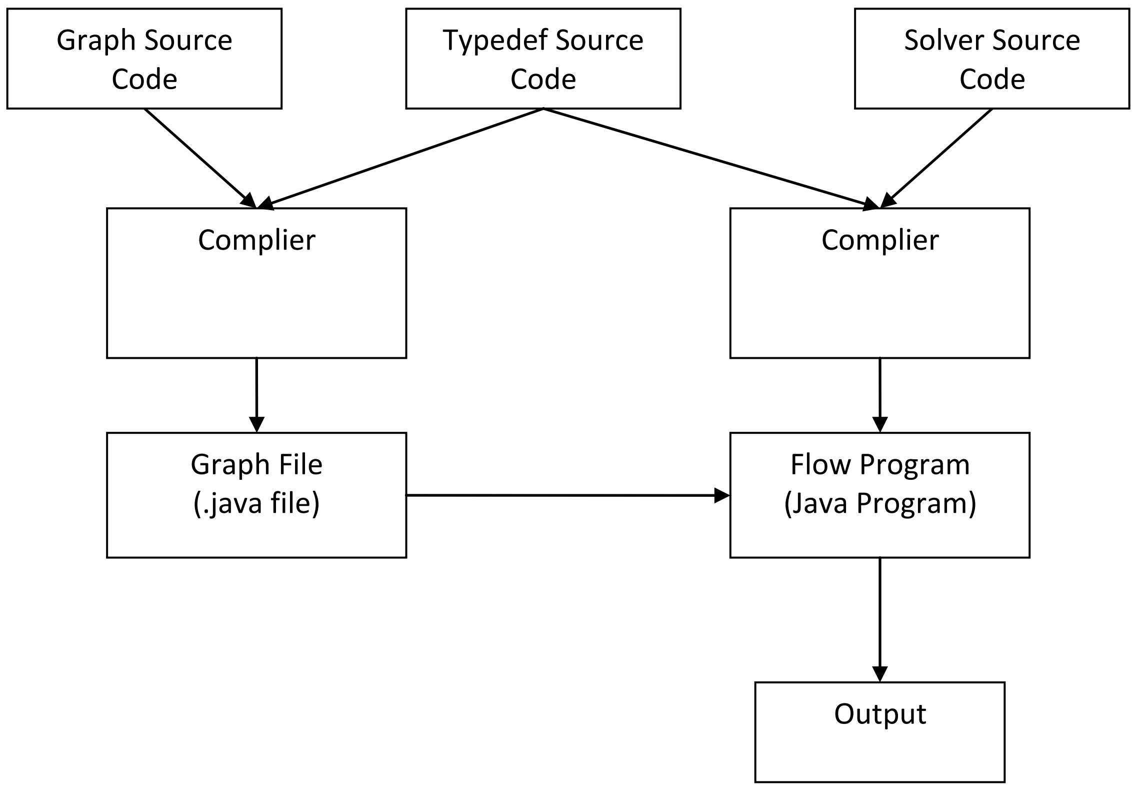
/* Solver */

```
//Specify the typedef
use 'dfa.flowt';

/* This function simulates the dfa specified above on its parameter input string */
function simulate(string input) {
   @current = Graph.START;
   print "The first current variable is set.";
```

```
    print "Entering the 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 the 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 = "aababbaab";
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";
}
```

Block Diagram


Graph Source Code
Typedef Source Code
Solver Source Code
Complier
Complier
Graph File (.java file)
Flow Program (Java Program)
Output