**The Flow Language: An Overview**

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

**Problem Domain**

The Flow language has two aspects, specifying a graph and specifying the computation of properties of said graph. Once the two aspects, graph specification and computation specification, are coded, they are plugged into one another and compiled into a program that creates the graph computationally, and then computes properties of the graph.

**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’s easy to apply a new solver to an existing graph to get additional information out of it. It’s also easy to use a tried-and-true solver on a brand new graph.

**Useful**: The Standard Library comes with solvers that compute lots of useful graph algorithms, including Dijkstra’s algorithm, Kruskal’s algorithm, and depth-first search. There are also libraries for other useful functions, such as the Producton Planning library with solvers for functions like min cost flow.

**Example Programs**

**Term Definition**

* **Grapher:** the code that generates the graph
* **Solver**: the code that does computation on the graph
* **Node:** the data type which represents the nodes of the graphs
* **Arc**: the connections between nodes

**File Types**

* **Typedef:** fundamental interface between the graph maker and the calculator.

*The typedef represents the context of the graph problem. For example, nodes and arcs used in Djikstra'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** (with a big G)**:** the specification for creating a graph (details are discussed below)
* **Solver:** a functional specification of the computation to be done on the graph

**Example Program #1**

//The typedef defines an interface across the Graph and the Calculator

/\* 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: \*/

//Create a depthN type node called 'a', label it with the special label root  
root a = depthN();

//Create two more depthN nodes  
b = depthN();  
c = depthN();

//Connect node 'a' to 'b', then node 'a' to 'c' with anonymous arcs  
depArc(a,b);  
depArc(a,c);

/\* Solver: \*/

function depth(depthN myNode) {

int depth = 0;  
 for depArc arc in myNode.arcs {

depth = max(depth,arc.endNode+1);

}  
 return depth;

}  
  
print depth(root);

**Example Program #2**

/\* Sample program: I use a graph to simulate a DFA. This DFA represents all strings of a's and b's with an even number of a's and an even number of b's. \*/  
  
/\* The typedef section \*/

/\* each dfa node has two attributes, and each path arc has one attribute \*/  
NODE dfa(string state, boolean isAccepting) START; //START is a required node label  
ARC path(string symbol);  
  
/\* Building the graph \*/

START s0 = dfa("s0", true); //state s0 is labeled as the START state  
s1 = dfa("s1", false);  
s2 = dfa("s2", false);   
s3 = dfa("s3", false);

/\* path definition: source, then destination, then attributes \*/  
path(s0, s1, "a");  
path(s1, s0, "a");   
path(s1, s2, "b");   
path(s2, s1, "b");   
path(s2, s3, "a");   
path(s3, s2, "a");   
path(s3, s0, "b");   
path(s0, s3, "b");   
  
/\* Solver Section \*/

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

function simulate(string input) {   
 dfa current = START;   
 for (int i = 0; i < input.length; i++) {   
 //find looks through all arcs and returns a list of all meeting the conditions  
 path next = find(ARCS, startNode==current, symbol==input[i])[0];  
 current = next.endNode;   
 }   
 if (current.isAccepting) return true;  
 else return false;   
}   
  
/\* All code not within a function definition will execute every time \*/

string input = "aababbaab";  
print "Input is '" + input + "'\n";   
  
if (simulate(input)) print "Input is accepted.\n";   
else print "Input is not accepted.\n";

**Block Diagram**

