DiGr

Directed Graph Processing

I. Team Members

Bryan Oemler (Team Leader)
Ari Golub
Joshua Lopez
Dennis V. Perepelitsa

II. Language Description

We present DiGr (pronounced to rhyme with "tiger"), a compiled, imperative, object oriented language designed to easily create, process and modify directed graphs. Directed graphs are a simple yet flexible idea which show up in everything from electrical networks and computer science data structures to gaming.

We propose a language in which nodes and edges are the most natural objects. DiGr's syntax allows for the creation of nodes, edges, and entire graph structures with small, concise statements. In DiGr, the user can write the minimum amount of information needed to define the digraph, and the compiler will intelligently fill in the rest of the details. We describe this in more detail in section V. Each node and directed edge efficiently stores any additional amount of user information, allowing for a wide variety of user applications. Where possible, DiGr tries to hide implementation details from the user: for example, undirected graphs are compiled as a special type of directed graph.

In DiGr, it is also easy to crawl and manipulate digraphs. "Crawls" are a special type of function defined in a way convenient for depth-first, breadth-first, or any other type of user-defined traversals of digraphs. That is, the most primitive function in the language is a recursive one that moves from parent to children nodes. Crawls are general enough to be useful in many graph-related applications, but narrowly defined enough to let the user do a lot while writing a little.

When given a start node and a "predicate" by the user, crawls use an internal queue to move through and examine or modify a directed graph. The user defines which action, if any, the crawl takes at a given node. The predicate guides graph traversal by determining the structure of the queue at each step. In the three variants of a depth first search, for example, the predicate determines the priority in which children nodes are visited. In a conditional path traversal, the predicate maintains a queue with only node in it. To aid in predicate programming, lightweight, SQL-style conditional statements can be used to select subsets of children nodes.

III. Problems Which DiGr Can Solve

DiGr can be used to solve a number of problems, both in and outside the realm of graph theory:

-Finding the best route between points based on various criteria (distance, cost, time required).
-Building search trees for fast storage and look-up of data (contacts lists, dictionary definitions, computer process trees)
-DiGr can also be used to implement finite state machines (and, by extension, regular expressions)

IV. Representative Programs

1. Shipping Industry

A sample program that would be perfect for our language is a modeling of a shipping industry. A main concern in shipping is getting the products to their destination in as economical a fashion as possible, be it the economy of time, money, or some other factor. Thus it is very important to have an easy yet sophisticated model of the shipping lanes, factories, and destinations involved. This type of application is perfect for DiGr.

The factories and destinations are represented as nodes and the shipping lanes that connect them are the edges. The edges would have a weight, or importance, that could be based on a number of factors including distance, frequency travelled, difficulty of transportation etc. When shipping lanes are temporarily disabled, say, due to weather, those edges would be represented as “broken”; the connection would still exist, but it would be inactive. And as you will see from our syntax, adding a new factory and connecting it to existing shipping lines is a trivial procedure.

2. Traveling Salesman/Graph Coloring

This application obviously lends itself to the “travelling salesman” problem. What makes DiGr so useful is that with the addition of predicates, the same traversal function could return widely different results with the change of a snippet of code unrelated to the function. For example, a function traverse(predicate P) could be used with two different predicates-- ‘take the edge with less financial cost’ or ‘take the edge of shorter distance’-- and return two very different routes. Or say each factory has one manager, and managers can only manage factories within one shipping lane of each other, DiGr could be used to easily implement a graph coloring procedure that would ensure this managerial property holds for all factories.

V. Language Syntax

Graphs

You can designate two objects when creating a graph: an array of nodes and an array of
edges. In this way, you can simply reference them by index value rather than name. For example, with an array of two nodes *points[2]* you can connect point one to point two with the simple syntax:

```
      points[] |1 -- 2 |!
      points[] |1 -> 2|!
```

‘|’ is the node assignment character; ‘->’ is a directed ege, ‘--’ is an undirected edge. ‘ ‘!’ is the line terminating character.

Assignments can be nested with parentheses.

```
      points[] |1 -> (2 -> (3,4)),5|!
```

Additionally, you can create an array of edges with properties. *WeightedEdges[5]* is an array of edges. The syntax for using these edges in a graph is as follows:

```
     points[] - weightedEdges[] |0 -a- 2; 1 -b -3; 3 -c- 4; 4 -d- 5; 5 -e-
0|!
```

Note that edges arrays are indexed by characters.

Predicates

The syntax used in predicates is considerably more declarative. The logic will be described in plain English terms, similar to SQL. Qualifiers like *greater than* and *less than* will be used to determine queuing order. For more complex values for nodes or edges, such as string values, the users define the hierarchy with a syntax similar to graph assignment:

```
     nodeValue | ‘a’ -> ‘b’ -> ‘c’ -> ‘d’|! // ‘a’ here is the ‘largest’
```

`value`

When crawls are executed, the predicate will be evaluated at each node. The result of the predicate is a stack with the nodes in desired order of processing.

Sample Program

```
//Define the entire tree in one line
binTree = []|8 -> (4 -> (2 -> (1,3),6 -> (5,7)),12 -> (10 -> (9,11),14 ->
(13,15)))|!

//Define a predicate for in-order traversal.
predicate inOrder{
      queue.add(FIND CHILDREN SMALLER this.value)!
      queue.add(THIS)!
      queue.add(FIND CHILDREN LARGER this.value)!
}

//Crawl the structure using the predicate and print out the node values
crawl (binTree[0]; inOrder;){
      print("This node's value is "+this.value)!
}
```

Output:

```
This node's value is 1
This node's value is 2
This node's value is 3
This node's value is 4
This node's value is 5
This node's value is 6
This node's value is 7
This node's value is 8
This node's value is 9
This node's value is 10
This node's value is 11
This node's value is 12
This node's value is 13
This node's value is 14
This node's value is 15
```