GOBLAN - Graph OBject LANguage

Uni	Name	Role
yk2553	Yunsung Kim	Project Manager
sl3368	Sameer Lal	Verification & Validation
jw3046	Jee Hyun Wang	Language Guru
dl2997	Da Liu	System Architect
sjg2174	Sean Garvey	System Architect

1. Motivation

Graphs appear naturally in problems of various domains such as physics, chemistry, sociology, linguistics etc. for the way they intuitively express entities and the complex relationships among them. Especially in the fields of computer science and statistics, graphs are used to represent data structures or social networks. Graphs can be used to analyze the organization of communities or even information diffusion. One of our interests is their use in probabilistic modelling. Bayesian statisticians use graphical models as a formalism for describing the relationship between events. This involves using graphs to represent distributions and formulas which ultimately generate data that is observed.

A method for exact inference in graphical models is the Junction Tree Algorithm (JTA). JTA is an algorithm that is based on the node connectivity in a graphical model. Each node passes to its neighboring nodes the "messages" that encode the partial information about the posterior distribution, and the neighbors collect these messages, run a function to create its own message, and pass it to their neighbors down the stream. The messages received by the last node in the stream is then used to reconstruct the desired answer.

We see two components of importance with this type of algorithm: the ability to (1) exchange messages between nodes, and (2) compute functions on the node-level using those messages to update the data on each node. In fact, this is exactly the intuition behind many graph algorithms. In a shortest path search (Dijkstra's algorithm), nodes pass to their neighbors distance vectors that represent the current distance estimates, which the neighbors use to update their own distance vectors. In a Depth First Search (DFS), every node receives search results from its children, examines those results, checks its own value if necessary, and passes its search result to the parent. Both of these examples fall into the paradigm of message passing.

Representing graphs and nodes in several common languages can be tedious and time consuming. As a result, we propose GOBLAN, a graph object language. This language focuses, not on manipulating the graph as a whole (with the use of adjacency matrices), but rather on exchanging data (messages) among individual nodes and using this data to update the state/data at each node. Our goal with this project is to develop a domain specific language which enables developers to work with graph based structures and algorithms with more ease (and speed hopefully!) by thinking in terms of the operations of each node, which we expect to provide a new methodology for graph programming.

2. Goals

- Intuitiveness and clarity: designing clear syntax and keywords that intuitively illustrate the operations of a node.
- Flexibility: Allowing multiple different types of algorithms for graph structures to be implemented with certain specifications
- Simplicity: designing a language that allow users to construct complex graph networks and their dynamics in terms of simple, node-level operations.

3. Language Description

GOBLAN is a graph manipulation language that represents a graph as multiple node objects, each of which conceptually consists of 3 parts - neighbors, data attributes, and object functions which update these attributes. There are two types of such object functions. A “synchronous” function is executed upon receiving a message from a neighboring node. An “asynchronous” function can be called independently from the exchange of packets. These two types of object function calls can be combined to construct an algorithm. We intend for the language to compile to Java.

In terms of syntax, GOBLAN combines the syntactic merits of AWK, Python, and C to give users the mixed flexibility of imperative programming, block definition, and object-oriented programming. Each GOBLAN script is broken down into “blocks.” There are 3 types of blocks: function definition blocks, main blocks (both similar to those in Python), and a graph definition block. The definition of a graph is done in three sections. The “data” section declares attributes possessed by each node, the “do” section defines the synchronous object function, and the “catch” section defines the asynchronous object function. In the definition of both types of object functions, the “pass” keyword is used to trigger the transfer of a package.

4. Syntax

Although the language reference manual will have the complete details of the syntax and grammar, following is a brief description of the keywords and how they are used.

3.1 Block Declaration Keywords

Keywords	*Description*
fun *fun_name* (type arg1, type arg2, …)	defines an external function
graph *graph_name*	declares a graph definition
main	the main script for execution

3.2 Array

Keywords	*Description*
len	get the length of the array
remove	remove the element in the array
add	add element to the end of array
min	output the minimum

3.2 Function Definition / Main Block Keywords (followed by keyword *graph*)

Keywords	*Description*
run	execute the asynchronous object function

3.3 Graph Definition Keywords (followed by the keyword *fun*)

Keywords	*Description*
data	begins the definition of graph attributes
do (*type arg1*, *type arg2*, …)	begins the definition of a node operation and the arguments used for the function
pass *packet* to {node|all}	triggers the node to forward the specified packet to a neighbor / all neighbors.
catch	definition of an asynchronous function
self	self-referring keyword for a node
pack	keyword in the *catch* statement that refers to the packet being passed
parents	The parents of a node in a directed graph
children	The set of nodes which are children of the current node in a directed graph
neighbors	All connecting nodes in an undirected graph

5. Example: Building a graph and finding the shortest path between 2 nodes

```
fun len(list l):
      int i = 0;
      for x in l{i = i + 1;}
      return x

graph DijkstraGraph:
      data{
            int dist = infinity;
            Node prev;
      }
      do(list l){
            pkt = {sender:self, dist: self.dist};
            pass ptk to all;
            l.remove(self);
      }
      catch{
            if (pack.dist + self.edge(pack.sender) < self.dist){
                  self.dist = pack.dist + self.edge(pack.sender);
                  self.prev = pack.sender;
            }
      }

main:
      DijkstraGraph G = read_graph(max_graph.json)
      l = [n for n in G.nodes];
      next = G[0];
      dest = G[len(G.nodes)-1]
      while (l.len > 0){
            next = min(l, key=dist);
            run next(l);
      }
      while (dest != G[0]){
            print dest;
            dest = dest.prev;
      }
```