The GAL Programming Language:

A rapid prototyping language for graph algorithms

Athar Abdul-Quader
ama2115@columbia.edu

Shepard Saltzman
sms2195@columbia.edu

Albert Winters
ajw2124@columbia.edu

Oren B. Yeshua
oby1@columbia.edu

1 Introduction

A graph G consists of a set of vertices V and a set of edges E each of which joins two of the vertices. This simple abstraction serves as a model for a multitude of real world systems and leads to a wide variety of useful and elegant algorithms for solving problems in those systems. However, despite the elegance of the formalism, implementing graph algorithms in a general purpose programming language can become quite cumbersome. Programmers must first make decisions about how to represent graphs, edges, and vertices, and then attempt to translate the algorithm into the desired language.

2 Audience

Students and researchers studying algorithmic theory should find GAL exceptionally useful in allowing them to quickly implement and experiment with the techniques they are investigating. GAL allows students to focus on algorithm concepts without worrying about time-consuming implementation details - helping to foster understanding of the algorithm as a whole, and perhaps facilitating the discovery of improvements where possible. GAL is also well suited for developers looking to quickly include graphs and graph algorithms in their software while avoiding the hassle of choosing an appropriate API and learning to use it.

3 Related work

While there are many graph and network algorithm APIs, like jGABL[*] and boost[†], they are all based on general purpose programming languages. As such, algorithms implemented using those APIs are encumbered with the syntax and nuances of the language, making them difficult to read and maintain. We have been unable to find a programming language developed specifically for computing with graphs.

[*]http://www.math.tu-berlin.de/jGABL/
[†]http://www.boost.org/libs/graph/doc/

4 Goals

4.1 Intuitive

The GAL language is terse and uncluttered by a myriad of non-essential symbols. GAL syntax closely mirrors popular pseudocode formats found in the algorithm literature making GAL code both easy to develop and intuitive to understand.

4.2 Concise

GAL programs should be concise in comparison to their counterparts in other languages. Because GAL is designed specifically for graph algorithms, it can provide significant LOC savings over the standard general purpose programming languages (both imperative and functional) when working with graphs. Built in data structures, operators, and functions for working with graphs and sets facilitate this goal.

4.3 Portable

The Java code produced by our GAL compiler can be quickly integrated into any Java application, providing the flexibility, scalability, and cross platform support that comes from using the Java language. Furthermore, GAL uses a simple set of primitives and built in functions that can be implemented in any general purpose programming language. While we will provide an implementation in Java, GAL compilers can be written for your preffered target language based on the GAL specification. Learning various APIs for different languages is time consuming and unnecesary.

4.4 Efficient

GAL will use appropriate data structures and algorithms in its internal representation and manipulation of graphs, sets, and queues in order to free the programmer from dealing with such issues. The focus of GAL is on rapid prototyping, so the core concern is to keep computationally efficient algorithms running efficiently when implemented in GAL. For fine tuning of constant factors, the code may be tightened in the target language.

5 Features

5.1 Data types

GAL includes graphs, sets, and queues as built-in types along with the familiar numbers, constants, strings, and booleans. The language is weakly typed for added flexibility and readability.

5.2 Control structures

Simple and familiar program flow control mechanisms like `while` and `for` loops and `if/else` statements are provided. Some language specific control structures are the `foreach` keyword as well as indentation to specify scope.

5.3 Comments

The traditional '//' signifies a single line comment, enabling a natural and readable embedding of annotation alongside the code.

5.4 A simple example

To get a sense of what a basic GAL program would look like, what follows is a depth-first traversal algorithm as it might be written in GAL.

```
DFS(G)
   foreach (u in G.V)
      u.visited <- FALSE          // initialize the vertices

   foreach (u in G.V)             // for all
      if (u.visited = FALSE)      // unconnected components of G
         DF-VISIT(u)              // begin traversal

DFS-VISIT(u)
   print(u)                       // output vertex info to the terminal
   u.visited <- TRUE              // mark node as visited
   foreach (e in u.edges)
      if (e.head.visited = FALSE)
         DFS-VISIT(e.head)        // recurse

MAIN()
   graph G                        // declare a graph
   G.V = {1,2,3,4,5}              // add vertices
   G.E = {(1,2),(1,3),(2,3)       // add edges
   G.E += {(4,5)}                 // add more edges
   DFS(G)                         // traverse
```

The simple program above demonstrates many of our GAL's features. Graphs, vertices, and edges are all basic types, but additional fields can be added to them on the fly as is the case with the `visited` field used to tag vertices in the example. Recursion is supported as in the call to `DFS-VISIT` and comments nicely annotate the code. The basic types are also printable for debugging or rapid prototyping purposes.

6 Summary

Programmers seeking to utilize graph algorithms in their software encounter a paradigm shift in which they must take the algorithms from the theoretical perspective in which they were discovered, and convert them into a functioning computer program. It is the aim of GAL to make this transition as simple and natural as possible. Algorithms implemented in GAL are human-readable, which allows this language to be used for teaching purposes as well as general graph algorithmic development. We hope our GAL will become a useful tool for studying, prototyping and implementing graph algorithms.