YAGL: Yet Another Graph Language
Proposal

Anthony Alvarez (aea2161), David Ding (dwd2112)
Columbia University

July 11, 2016

Contents

1 Introduction

We seek to make a language, YAGL (Yet Another Graph Language), which allows users to interact with graphs in a manner similar to the language of mathematical proofs. This should allow a person with a theoretical understanding of graphs to engage with them in an intuitive practical way.

With YAGL, users will be able to easily create graphs and add vertices and edges, and associate arbitrary attributes with them (for example, colors with vertices and weights with edges). Common graph algorithms will be able to be written cleanly and concisely, as if they came out of the classic Algorithms textbook by CLRS.

2 Language Design and Syntax

This section provides a rough idea of the design of YAGL, and is subject to change. Final language design and syntax will be provided in the language reference manual.

2.1 Comments

YAGL will use Python-style comments for single line, and will not have a special syntax for multi-line comments. Example:

```
# This line is a comment.
```

2.2 Data Types

Basic types:

Type	Explanation
int	integer
float	floating point number
string	sequence of characters

YAGL also implements a notion of infinity and negative inifinty using the keywords INF and -INF respectively. INF is greater than every int or float and is equal to INF. -INF is less than every int or float and is equal to -INF.

YAGL implements constants True and False which evaluate to 1 and 0 respectively. These constants add syntactic sugar to algorithms though they are directly replaceable with ints.

Additionally we implement a Null which can take the place of int, float, or string. Null comparisons are always false. To detect nulls YAGL implements an isNull built in function.

Collection types:

Type	Explanation
list	a list of values, all of the same type
map	a map of from keys (alphanumeric strings) to values of arbitrary types
set	a set of values, all of the same type

Graph types:

Type	Explanation
Graph	an undirected graph
Digraph	a directed graph

2.2.1 Elaboration on Graphs

A graph has vertices and edges. Fundamentally, the edges in a graph are either all undirected or all directed, so we have the graph types Graph and Digraph. Vertices will be accessed by labels, and edges will be accessed via two vertex labels.

One will be able to associate any vertex or any edge with any number of attributes. These attributes will be keyed by alphanumeric, and have a value of arbitrary type. In particular, note that edge weights are not a fundamental attribute within the language, but one will be able to easily define an edge weight via, say, an attribute with name 'w' and either a float or int value.
Vertices will have the immutable attribute 'label', and edges will have the immutable attributes 'orig' and 'dest', representing the origin and destination vertices. These attributes will be defined when the vertex or edge is created.

2.3 Operators

2.3.1 Basic Operators

Category	Data Type	Operator	Explanation
Comparison	int, float, string	==, !=, >, <, <=, >=	Act the same as C++ operators.
Computation	int, float	+,-,*,/,%	Act the same as C++ operators.
Computation & Assignment	int, float	+=,-=,*=,/=	Act the same as C++ operators.
Concatenation	string	+	Concatenates two strings.
Concatenation & Assignment	string	+=	Concatenates right hand side string to original and assigns.
Boolean	int, float	!	0 maps to 1, and all other values map to 0.
Boolean	int, float	AND	1 if both values are nonzero, and 0 otherwise.
Boolean	int, float	OR	0 if both values are zero, and 1 otherwise.

2.3.2 Collection Operators

Category	Data Type	Operator	Explanation
Comparison	list, map, set	==, !=	If all values in all indices/keys are equal then == returns True else != returns True.
Contains	list, map, set	in	Returns if an item exists in the list,map, or set
Concatenation	list, set	+	Concatenates two lists or sets. On sets removes duplicates.
Concatenation, Assignment	list, set	+=	Concatenates right hand side list to original and assigns
Removal	set, map	-	Removes all items in the right hand set, or map (by key), from the left hand set if they exist
Removal, Assignment	set, map	-=	Removes all items from in the right hand set, or map (by key), from the left hand set if they exist and assigns back to left hand side set.
Access	list	[i]	Access the ith element of a list (zero-indexed)
Access	map	myMap.literal or myMap["literal'] or myMap[variable]	YAGL supports multiple map access methods for ease of use. myMap.literal is identically equivalent to myMap["literal']. The bracket notation must be used if accessing using a a variable.

2.3.3 Graph Operators

Category	Data Type	Operator	Explanation
Comparison	Graph, Digraph	==, !=	If all attributes in all vertices and edges are equal then == returns True else != returns True.
Concatenation	Graph, Digraph	+	Concatenates two graphs together into a single graph. Note, names of vertices in both graphs must be distinct to concatenate them.
Concatenation & Assignment	Graph, Digraph	+=	Concatenates two graphs together into a single graph. Note, names of vertices in both graphs must be distinct to concatenate them.

2.4 Built-in Functions

Code	Explanation
print(arg1,arg2,...)	print out a comma separated list of variables and a new line character
size(iterable)	return the size of iterable
isEmpty(iterable)	return size(iterable) $>$ 0
isNull(arg)	returns 1 if the arg is Null 0 otherwise
enqueue(list, elem)	add an element to the end of a list
dequeue(list)	remove the first element of a list and return it
push(list, elem)	add an element to the end of a list
pop(list)	remove the last element of a list and return it
adj(graph, v)	return the set of vertices in a graph adjacent to v

2.5 Control Flow

Code	Explanation

if(condition)
  # If condition code
else
  # Else condition code
	if-else block

while(condition) # While condition code	while loop
forEach(item, iterable) # code operating on each item of iterable	simple for loop
forComponentValue(key, value, iterable) # code operating on each key-value pair	enumerating for loop

2.6 Function Definition

Functions will be defined using the keyword def and the syntax below and the return keyword will return any output of the function

```
def myFunction( Arg1, Arg2 )
    # The code of my function
    return( 'This is the return value of myFunction()' )
```

3 Sample Code

```
## Simple collection manipulation

# Define a List
a = []
push( a, 1 )
push( a, 2 )
a == [1,2]
>> 1
pop( a )
>> 2
a
>> [1]
enqueue( a, 3 )
a
>> [1,3]
dequeue(a)
>> 1

# Define a map
c = {||}
c.key1 = 'Value1'
c.key2 = [ 0, 1, 3 ]
c.key3 = 10
print( size( c ) )
>> 3

# Define a set
d = {}
forEach( elem, c.key2 )
    d += elem
print( d )
>> {0,1,3}
d += 3
print( d )
>> {0,1,3}
d -= { 0, 1, 3, 4 }
print( isEmpty( d ) )
>> 1

## Build and examine a basic Graph
G = Graph()
G.V += 'a'
G.V += {'b','c'}
forEach( v, G.V )
    G.E += [v,v]
    G.E[v,v].weight = 1
G.E += [ 'b','c' ]
print( G.E[ 'b', 'c' ].weight )
```

```
>> Null

#Create a copy of G as a digraph
D = Digraph()
forComponentValue( label, attributes, G.V )
    D.V += label
    forComponentValue( key, value, attributes )
        D.V[key] = value
forEach( edge, G.E )
    D.E += [ edge.orig.label, edge.dest.label ]
    D.E += [ edge.dest.label, edge.orig.label ]
    forComponentValue( key, value, edge )
        D.E[ edge.orig.label, edge.dest.label ][ key ] = value
        D.E[ edge.dest.label, edge.orig.label ][ key ] = value

###############################################
# BFS takes as arguments a graph or digraph G #
# and s is a string representing the label of #
# the desired root vertex                     #
###############################################
def BFS( G, s )
    #Takes a graph G and a label for a start node s
    forEach( v, G.V )
        if( v.label == s )
            v.color = 'gray'
            v.d = 0
            v.parent = NULL
        else
            v.color = 'white'
            v.d = INF
            v.parent = NULL
    queue = []
    enqueue( queue, G.V[s] )
    while( !isEmpty( queue ) )
        u = dequeue( queue )
        forEach( v, adj( G, u ) )
            if v.color == 'white'
                v.color = 'gray'
                v.d = u.d + 1
                v.parent = u
                enqueue( queue, u )
        u.color = 'black'
    return( G )

###############################################
# Relax is a helper function for BellmanFord  #
# it takes as arguments a graph or digraph G  #
# and an edge e in that graph                 #
###############################################
def Relax( G, e )
```

```
    #If the distance of is more than the weight of the edge u->v and the distance on u
    v = e.dest
    u = e.orig
    if( v.d > u.d + e.weight )
        v.d = u.d + e.weight
        v.parent = u
    return()

###############################################
# The BellmanFord algorithm takes a graph or  #
# a digraph G and a label for a source s and  #
# returns True if there is a negative weight  #
# cycle that is reachable from s              #
#                                             #
# returns False if there is not a negative    #
# weight cycle that is reachable from s       #
###############################################
def BellmanFord( G, s )
    ret = True
    forEach( v, G.V )
        v.d = INF
        v.parent = NULL
    G.V[s].d = 0
    forEach( i, range( 0, size( G.V ) ) )
        forEach( edge, G.E )
            Relax( G, edge )
    forEach( edge, G.E )
        if( edge.dest.d > edge.orig.d + edge.w )
            ret = False
    return( ret )
```