d.o.t.s.
A graph language.

Hosanna Fuller (hjf2106) — Manager
Rachel Gordon (rcg2130) — Language Guru
Yumeng Liao (yl2908) — Tester
Adam Incera (aji2112) — System Architect

September 2015

Contents

1 Introduction and Motivation

Graphs are a powerful and versatile data structure used across many languages to help visually organize and manipulate data. Many languages do not provide out-of-the-box support for data structures and methods useful in solving graph problems. Because of this, programmers end up spending unnecessary time and energy implementing these critical structures themselves. As a result, graph implementations often widely vary in efficiency and modularity. The goal of `d.o.t.s` is to provide an out-of-the-box graph framework so that users can focus on creating the algorithms needed to solve their problems, rather than getting bogged down in implementation issues. With `d.o.t.s`, users can comprehensively solve a wide variety of graph-based problems. Some example problems include: expressing network relationships such as a series of interconnected routers with edge costs, representing decision trees in probability, and running analyses on propositional models.

2 Language Design and Syntax

The strict typing and control flow in d.o.t.s. is reminiscent of C and Java, but overall the language is intended to be used more as a scripting language, where the user builds their graphs quickly using the intuitive node and edge operators and then performs operations on the structures.

The d.o.t.s. compiler compiles code written in d.o.t.s. into C binary executables.

Note: In the following sections, the word "graph" is sometimes used to denote a data structure and sometimes to denote the abstract structure from computer science and mathematics:

> A graph data structure consists of a finite (and possibly mutable) set of nodes or vertices, together with a set of ordered pairs of these nodes (or, in some cases, a set of unordered pairs). These pairs are known as edges or arcs. As in mathematics, an edge (x,y) is said to point or go from x to y. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references.
>
> A graph data structure may also associate to each edge some edge value, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.). [1]

For the sake of clarity, from this point forward we will refer to the language-specific data structure using the lowercase "graph" and the mathematical concept using the uppercase "Graph."

2.1 Comments

Syntax	Comment Style
* *code* *\	multi-line comment
#	single-line comment

Table 1: Comment Styles

2.2 Built-in Data Types

d.o.t.s. comes with four basic types: `int, bool, float,` and `string`. Each of these basic types can be used as raw values with no prior declaration of variables or can be assigned as the values of variables. d.o.t.s. also provides two built-in data types, `node` and `graph`, which provide the basis for algorithms written

[1] `https://en.wikipedia.org/wiki/Graph_(abstract_data_type)`

in d.o.t.s.. The built-in collections are: `list, dict,` and `pqueue`. d.o.t.s. is a strictly-typed language, meaning that the types of all variables must be declared at the same time that the variable is declared. The exception to this rule is that the types of node values do not need to be declared. Programmers can insert values of any type in the `value` field of a `node`, and at compile time, the compiler will mark their data type.

In addition to these data types, d.o.t.s. also includes the value `null`, which represents the absence of value for any data type.

Data Type	Fields
int	
float	
string	
bool	
list	
dict	
pqueue	
node	value, in, out, marked
graph	node_list

Table 2: Built-in Data Types

Explanation of Built-in Types

The data types which underpin d.o.t.s. and give it its advantage in the Graph domain over languages such as C are `node` and `graph`. From the get-go any programmer using d.o.t.s. can use these data types to quickly build Graphs without the need to waste time creating these data structures from scratch.

A `node` object represents a single vertex in a Graph, whereas a `graph` object represents a collection of graphs (which can be empty). Nodes and graphs demonstrate the square-rectangle relationship, in that a node is a graph, but a graph is not a node.

Recursive definition of graph objects:

- An empty graph is a graph.
- A node is a graph.
- A graph added to a graph is a graph.

A graph contains only the field `node_list`, which is a list of all node objects contained within the graph.

A node contains the fields `id, value, in, out, marked`. The `id` field is a unique identifier of the node that is set by the compiler. The `value` field can be an object of any type, and simply represents some value that the node contains. One possible use of the value field is to allow users to assign a more semantic meaning to nodes (ex. setting the value to the name of a city). The `in` field is a dict mapping nodes that the current node has edges into to weights. Similarly, the `out` field is a dict mapping nodes that have edges into the current node to weights. The keys of the two dicts are the unique id's of the nodes. An example of accessing the `in` and `out` dicts of a node can be seen in Listing 2. The bool field `marked` is intended for use in search algorithms to represent whether or not the node has already been seen.

Since node has an *is-a* relationship with graph, it also contains a `node_list` field, but this is set upon declaration to contain only the node itself, and cannot be altered by the user.

Nodes can be declared in two different ways. In the first, a the variable can be simply be declared with the `node` keyword and a variable name. This creates a basic node with an empty value, in_list, and out_list. In the second manner, a node can be declared by giving it an initial value inside parentheses after the variable

name (as seen in line 11 of Listing 1). Alternatively, a declared variable can be initialized with the assignment operator "=" to any object of the type `node`.

Graphs can be declared as a variable name only or alternatively be assigned a value at declaration time. A graph can be assigned any expression that evaluates to something of the type `graph` (as seen in line 9 of Listing 1).

Explanation of Collections

Lists are declared using the keyword `list` and an indicator of the type of the list, as all objects in a list must be of the same type. Lists can be assigned by putting a comma-separated list of objects inside brackets, as seen in line 5 of Listing 1.

Dictionary objects in d.o.t.s. represent mappings from strings to objects; all keys must be of the type string, and all objects in a single dict must be of the same type. Dict objects are declared in a similar manner to lists, using the keyword `dict` and an indicator of the type that the dict maps to. Dicts can be assigned by putting a comma-separated list of (key:value) pairs inside curly braces, as seen in line 6 of Listing 1.

The `pqueue` object represents a priority queue in d.o.t.s. with the distinction that objects contained in the priority queue do not *themselves* need to be directly comparable. Instead, when objects are inserted into the priority queue, they are inserted along with a number that represents their value. As with lists and dicts, pqueues must be declared along with an indicator as to the type of object it contains. d.o.t.s. has no basic queue type. Instead, a basic queue can be simulated using a `pqueue` by using the same weight value for all inserted objects. Pqueues can only be assigned by setting them equal to another object of type `pqueue`.

```
1  int x = 12;
2  float y = 12.0;
3  string z = "12";
4
5  list<int> intList = [1, 2, 3];
6  dict<int> intDict = {"one" : 1, "two" : 2, "three" : 3};
7  pqueue<node> nodeQueue;
8
9  graph g1;
10 graph g2 = g1;
11 node x;
12 node y("nyc");
```

Listing 1: Declaration of built-in data types.

2.3 String, Int, Float Operators

Category	Data Type	Operator	Explanation
comparison	int, float, string	<, >, <=, >=, !=, ==	Operate in the same way as languages such as C/C++, with the exception that string equality compares the *value* contained in the string.
computation	int, float	+, -, *, /, %	Operate in the same way as languages such as C/C++.
concatenation	string	+	String concatenation operator

Table 3: Comparison Operators

d.o.t.s. provides an infinity value for both floats and ints: `float.INF` and `int.INF`. The operators perform a little differently for these values. As the primary use of infinity in graph problems is to define edge weight and not to perform mathematical calculations, the computation operators return `null` whenever infinity is an operand.

For comparison operators, `INF` values are greater than all non-null non-infinity values and equal to other infinity values of the same type (i.e. int or float). Defining the comparison operators for `INF` values allows them to be used both as valid edge weights, and valid weights in priority queues, which can be useful for graph problems.

2.4 Node Operators

The node operators outlined in Table 4 are all binary operators which take a node object on the left-hand and right-hand sides of the operator.

Operator	Explanation
−−	Add undirected edge with no weights between the 2
−−>	Add directed edge from left node to right node with no weight
−−[*num*]	Add an undirected edge between 2 nodes with weight *num* in both directions
−−>[*num*]	Add a directed edge from the left node to the right node with weight *num*
[*num*]−−[*num*]	Add edge from left to right with the weight in the right set of brackets, and an edge from right to left with the weight in the left set of brackets
==	Returns whether the internal ids of 2 nodes match
!=	Returns whether the internal ids of 2 nodes do not match

Table 4: Node Operators

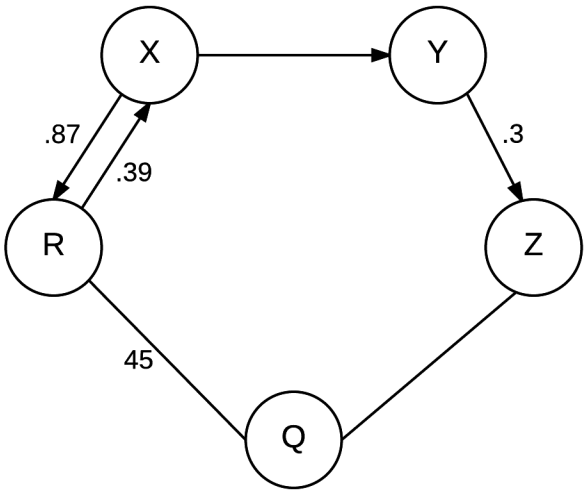

Figure 1: Example Graph showing nodes with different weights and edges.

```
node X, Y, Z, Q, R;

X --> Y;
Y -->[.3] Z;
Z -- Q;
Q --[45] R;
R [.87]--[.39] X;

R == Q; # returns false
R != Q; # returns true

\* accessing edge lists: *\
X.out[Y.id]; # == null
Y.out[Z.id]; # == .3
R.in[X.id]; # == .87
```

Listing 2: Shows the use of node operators that creates the graph in Figure 3.

2.5 Graph Operators

The graph operators outlined in Table 5 are all binary operators which take a graph object on the left-hand and right-hand sides of the operator.

Operator	Explanation
+	Returns a graph that contains all of the graphs in the left-hand and right-hand graph
+=	Adds the graph on the right-hand side of the operator to the graph on the left-hand side.
-=	Removes the graph on the right-hand side of the operator from the graph on the left-hand side.
==	Returns whether the two graphs contain the same nodes.

Table 5: Graph Operators

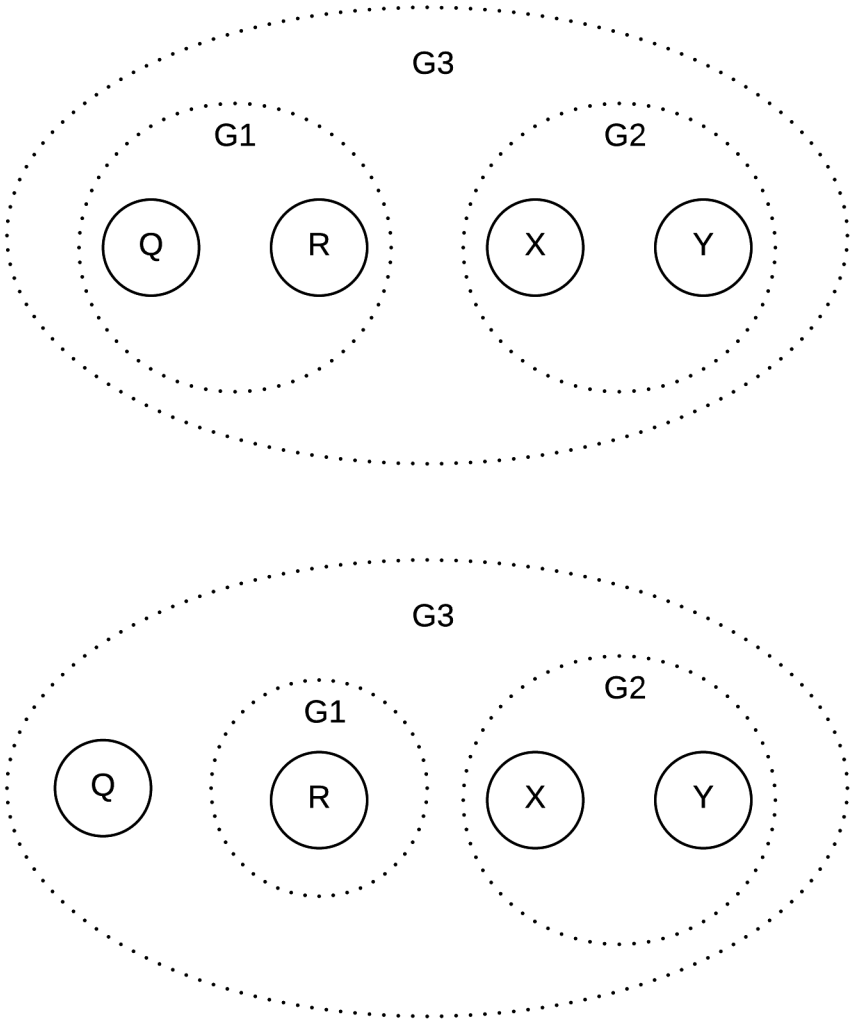

Figure 2: Example showing graphs and graph nesting. The bottom graph is the result of removing the node "Q" from the graph G1.

```
node X, Y, Q, R;
```

```
graph G1, G2, G3;
G2 = X + Y;
G1 = Q;
G1 += R;
G3 = G1 + G2; # result is the top graph
G1 -= Q; # result is now the bottom graph
```

Listing 3: Shows the use of graph operators that creates the top graph in Figure 2 and then alters it to the bottom graph shown.

2.6 Built-in Functions

Syntax	Explanation
print(x, ...)	prints to standard output the string representation of a list of comma-separated values or variables.
range(*int_lower*, *int_upper*)	returns a list of the integers from *int_lower* to *int_upper*, inclusive. The first argument can be ommitted, in which case 0 will be used as the default value of *int_lower*. The data type of both arguments must be int.
len(*iterable_var*)	returns the length of the iterable variable
isEmpty(*iterable_var*)	returns `len(iterable_var) > 0`
getNodeById(*node_id*)	returns the node object with the id *node_id*
node functions: mark(*bool_value*)	 sets the `marked` field of the node to the given bool value. If the argument is ommitted, *bool_value* defaults to `true`
pqueue functions: enqueue(*object*, *num_weight*) dequeue()	 adds the given object to the priority queue with the given weight returns and removes the object with the lowest weight from the queue

Table 6: Built-in Functions

```
list<int> x = range(1, 3);
list<int> y = range(3);

print("x: ", x, "\ny:", y);
\* prints out -->
  x: [1, 2, 3]
  y: [0, 1, 2, 3]
*\

node n;
n.mark();
print(n.mark); # prints --> true
print(n == getNodeById(n.id)) # prints --> true

node x, y, z;
```

```
pqueue q;
q.enqueue(x, 13);
q.enqueue(y, 2);
q.enqueue(z, 3);
print(q.dequeue()); # prints --> y.id
print(len(q)); # prints --> 2
print(isEmpty(q)); # prints --> false
```

Listing 4: Shows the use of built-in functions.

2.7 Control Flow

As Listing 5 includes example usage for each of the different types of control statements, this section omits a separate demonstration of their use.

	Explanation

if *condition* {
* code *\
}
else {
* code *\
}
	if else statement

while *condition* {
* code *\
}
	while loop

for *var_name* in *iterable_var* {
* code *\
}
 	
example:
for *node_var* in *graph_var* {
* code *\
}
	Iterates through all the elements of the iterable variable, assigning the current element to *var_name*.
	
	Iterates through all the nodes contained in *graph_var*, assigning the current node to the variable *node_var*.

Table 7: Control-flow Syntax

3 Sample Code

In this section, we demonstrate how a simple path-searching algorithm can be implemented using d.o.t.s.'s syntax. The end-goal of our project is to be able to implement more complex algorithms such as Djikstra's algorithm. But for the purposes of demonstration, we chose to show the Breadth-First Search algorithm, as its implementation makes use of each collection type and control-flow statement.

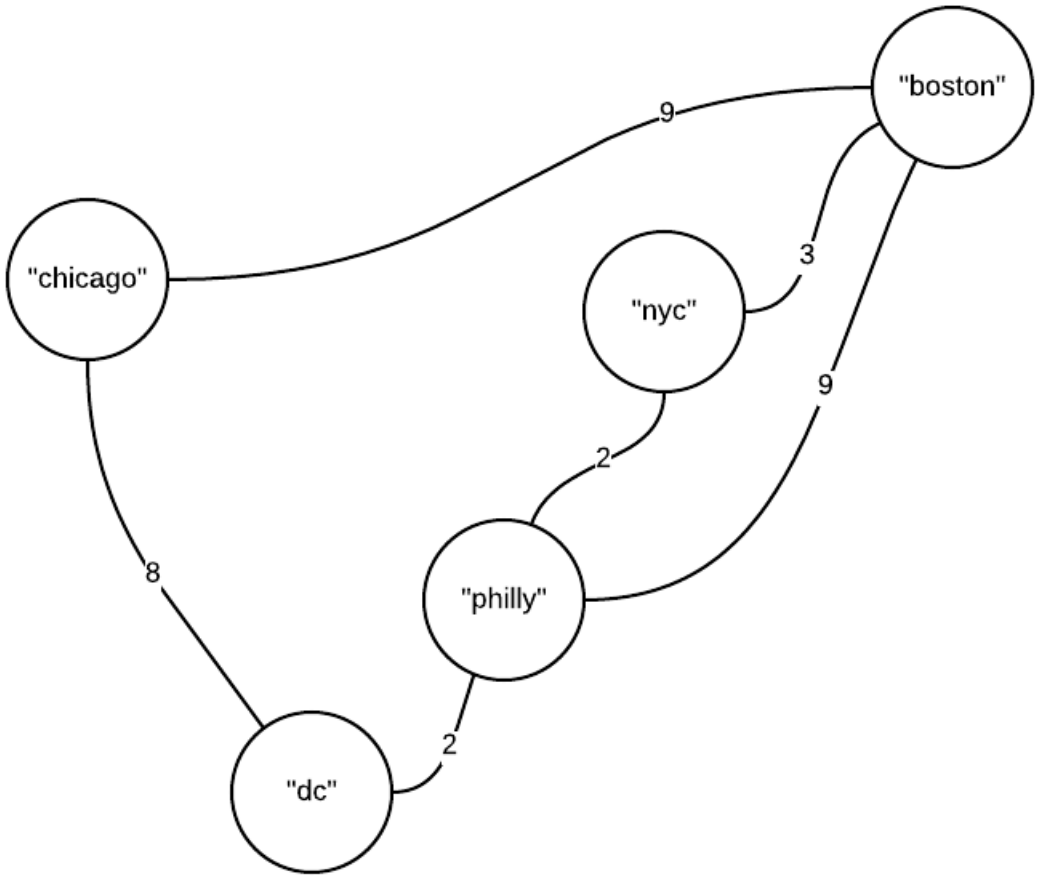

Figure 3: Visual of graph created by sample code.

```
\* Graph set-up *\
node x("dc"), y("chicago"), z("philly"), q("nyc"), r("boston");
x --[2] z;
z --[2] q;
q --[3] r;
z --[9] r;
x --[8] y;
y --[9] r;

graph g1 = x + y + q + r + y;
\* end Graph set-up *\

\* breadth-first search *\
node start = z;
node end = r;

node next;
pqueue<node> search;
dict<node> parent_list = {y.value : null};
```

```
y.mark();
search.enqueue(y, 0);
bool found = false;

while !isEmpty(search) {
 next = search.dequeue();

 if next == end {
  found = true;
  break;
 }

 list<string> children = next.out.keys;
 for c in children {
  node child = getNodeById(c);
  if !child.marked {
   parent_list[child.id] = next;
   child.mark();
   search.enqueue(child, 0);
  }
 }
}

if found {
 node cur = next;
 list<node> path = [cur];
 while parentList[cur.id] != null {
  cur = parentList[cur.id];
  path.add(cur);
 }
 print (path);
}
else {
 print ("path not found");
}

\* end breadth-first search *\

\* Djikstra's algorithm, calculate paths starting from "philly" and return a
   list of nodes with the order of traversal*\
node source = z;
pqueue<node> nodeSet;
dict<int> dist;
dict<node> parentList;

dist[source.id] = 0;
parentList[source.id] = null;

for vertex in g1 {
   if vertex != source {
      dist[vertex.id] = int.INF;
      parentList[vertex.id] = null;
   }
   nodeSet.enqueue(vertex, int.INF);
```

```
}

while !isEmpty(nodeSet) {
   node u = nodeSet.dequeue();
   for outNodeId in u.out {
      int altDist = dist[u.id] + u.out[outNodeId];
      if altDist < dist[outNodeId] {
         dist[outNodeId] = altDist;
         parentList[outNodeId] = u;
      }
   }
}
print(dist);

\* end Djikstra's *\
```

Listing 5: Path Searching