GRI: Interpreter of a dynamic language for GRaph algorithms

Sandeep Dasgupta
University Of Illinois at Urbana Champaign.
sdasgup3@illinois.edu

ABSTRACT

As graphical models are increasingly become popular in various fields, the domain experts often struggle to represent and compute on such models in a convenient and efficient way. In this project we develop a dynamically typed language which provides the desired convenience of representation of those models without loosing much on the efficiency of doing computation on them.

Keywords

Graph Algorithm, dynamically typed language, interpreter

1. INTRODUCTION

As graphical models are increasingly being used in various fields like biochemistry (genomics), electrical engineering (communication networks and coding theory), computer science (algorithms and computation) and operations research (scheduling), organizational structures, social networking, there is a need to represent and allow computation on them in a convenient and efficient way. This involves (but not limited to)

- Designing a language which provide an convenient interface to the programmer to program those models. This is essential so that even for domain experts who are not coding experts can code and reason about their implementation. Ease of interface could be due to:
  - Expressive power of the language representing those models.
  - Intuitive extensibility of the language.
  - Ability of the language to provide exploratory programming, where the user may experiment with different ideas (without dwelling much into the language syntax) before coming to a conclusive one.

  For the above reasons we are proposing a dynamically typed language (where a variable can bound to a value of any type) to represent the graphical models. Following are some of the benefits of a dynamically typed language:

  .

  - It's more concise - A lot of extraneous boilerplate code (related to type declarations, type casting logic) can be removed. Shorter code is marginally quicker to write, but more importantly it can be quicker to read and maintain (since we don't need to wade through many pages of code to get a grip on what is happening)
  - Dynamic typing is arguably more suitable for interactive, REPL-like programming for rapid prototyping, real-time debugging of running program instances.
  - Lack of compile time, meaning quicker turnaround.
  - Can pass variables/objects between routines/modules without having to know or declare their type.

- As programs in our language is going to be interpreted, we will be loosing performance w.r.t the compiled version of those programs. The reason we are making this trade off (of designing an interpreter as opposed to a compiler) is because our priority is to provide convenience to the programmer.

  Despite of the obvious reason of slowdown, we should strive not to loose much on the runtime performance. Designed language need to be efficient in the following sense.

  - Underlying design decisions including data structures need to be carefully crafted so that we should not loose performance because of bad design choices.
  - Implementation need to be scalable w.r.t the space/time requirements. This is important because most of the graph algorithm typically work on huge input sizes.

2. RELATED WORK

Our work in mostly inspired by the line of work by GUESS [4] and Graphal [2].

GUESS, a novel system for graph exploration that combines an interpreted language with a graphical front end that allows researchers to rapidly prototype and deploy new visualizations. GUESS also contains a novel, interactive interpreter that connects the language and interface in a way that facilities exploratory visualization tasks. They used a domain specific embedded language which provides all the advantages of Python with new graph specific operators, primitives, and shortcuts.

Graphal is an interpreter of a programming language that is mainly oriented to graph algorithms. There is a command line interpreter and a graphical integrated development environment. The IDE contains text editor for programmers, compilation and script output, advanced debugger and visualization window. The progress of the interpreted and debugged graph algorithm can be displayed in 3D scene.

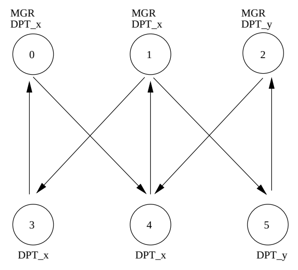

Figure 1: An example graph

```
         function main(argv) main() {
S1:        g = graph();
S2:        ...
S3:        g.setDirected(true);
S4:        ...
S5:        v0 = g.createVertex();
S6:        v0.__id = 0;
S7:        v0.__DPT_x = 1;
S8:        v0.__DPT_y = 0;
S9:        v0.__MGR = 1;
S10:       ...
S11:       v3 = g.createVertex();
S12:       v3.__id = 3;
S13:       v3.__DPT_x = 1;
S14:       v3.__DPT_y = 0;
S15:       v3.__MGR = 0;
S16:       ...
S17:       g.createEdge(v0,v3);
         }
```

Figure 2: Example code snippet to create a graph

Our language design is inspired by the above two work. But we additionally provided a number of built-in functions for supporting some basic computations on graphs. This not only help us getting convenient short hand notations to achieve those basic computation, but also we gain on performance due the fact that those basic tasks are now available in compiled version.

3. QUICK TOUR OF LANGUAGE

In this section we will provide some insights of designed language using some motivating examples.

EXAMPLE 1. *Figure1, shows a directed graph where the nodes represent the employees and the edges between them represents the email conversation from one employee to other. For example,* `Node 0` *represents an employee in* `DPT_x` *who is a Manager as well (this is attributed by the* `MGR` *tag). Similarly,* `Node 3` *represents a* `DPT_x` *employee. The edge between* `Node 3` *and* `Node 0` *represents the email conversation from employee* `Node 3` *to employee* `Node 0`*.*

Now let us first talk about the ways to represent this graph in our implementation. One way is as shown in 2.

At line `S1`*, a new graph variable is created.* `S3` *sets that the graph is directed.* `S5` *creates a node* `Node 0` *of this graph. As nodes and edges are the basic building block of a graph we have made them the first class objects which allow users to access them directly. Lines* `S6` *-* `S9` *sets various properties of the vertex. For example, it says that the vertex has* `__id = 0` *, it belongs to* `__DPT_x`*, does not belongs to* `__DPT_y` *and its* `__MGR` *property is true. Similar properties are set for vertex node with* `__id = 3` *(Lines* `S11 - S15`*). And finally a directed edge between them is created at line* `S17`*.*

```
         function main(argv) main() {
S1:        g = graph();
S2:        g.loadFromFile(argv[0]);
S3:        displayAdjMatrix(g.getAdjacencyMatrix(), g.getVertices());
S4:        ...
S5:        dpt_x_employee = g.getVertexSetWithProperty("__DPT_x", 1.0);
S6:        mgr_employee = g.getVertexSetWithProperty("__MGR", 1.0);
S7:        ...
S8:        /* Set of __DPT_x employees who are __MGR as well */
S9:        dpt_x_AND_mgr = dpt_x_employee.intersection(mgr_employee);
S10:       ...
S11:       println("Set of __DPT_x employees who are __MGR as well");
S12:       foreach(employee; dpt_x_AND_mgr) {
S13:         println(" " + employee.__id + " ");
S14:       }
S15:       ...
S16:       /* Set of __DPT_x employees who are NOT __MGR */
S17:       dpt_x_MINUS_mgr = dpt_x_employee.difference(mgr_employee);
S18:       ...
S19:       it = dpt_x_MINUS_mgr.iterator();
S20:       ...
S21:       println("Set of __DPT_x employees who are NOT __MGR");
S22:       while(it.hasNext()) {
S23:         employee = it.next();
S24:         println(" " + employee.__id + " ");
S25:       }
S26:       ...
S27:       // Email Exchanges from MGRs to non-MGR DPT_x employee
S28:       emailExchanges = mgr_employee -> dpt_x_MINUS_mgr;
S29:       ...
S30:       // Email Exchanges from non-MGR DPT_x employee to MGRs*/
S31:       emailExchanges = mgr_employee <- dpt_x_MINUS_mgr;
S32:       ...
S33:       // Email Exchanges between non-MGR DPT_x employee and MGRs*/
S34:       emailExchanges = mgr_employee <-> dpt_x_MINUS_mgr;
S35:       ...
         }
```

Figure 3: Example code snippet to create a graph

One thing to note here is that the methods like `graph()`, `setDirected()`, `createVertex()` *and* `createEdge()` *are all built-in compiled functions.*

Figure 3 shows another way to represent graph. It uses an input file to be fed to the interpreted program. The format of the input file is shown in Figure 4. Such an input file is read from the command line arguments and used to create a graph as shown in line `S2` *of Figure 3. At* `S3`*, we can see two built-in functions* `getAdjacencyMatrix` *&* `getVertices` *on graph which respectively gives an array of array representing the adjacency matrix representation of the graph and a set of vertices in the graph.* `displayAdjMatrix` *is just a method call, the definition of which is not shown for brevity. Now lets try to solve certain queries using the graph at Figure 1.*

In case we want to get all the nodes in the graph who are `__DPT_x` *employees, then the query to get all those nodes is shown at line* `S5`*. Similarly, the query at line* `S6`*, gives the nodes for which the property* `__MGR` *is set to true. One thing to note here that the return value of both the queries are sets which are amenable to set operations.*

For example, in case we want to get all the employees who are both department `__DPT_x` *employee and managers , we can get that using the set intersection as shown at line* `S9`*. Line* `S8`*, shows the multiline comment we support. Line* `S12` *shows a way to iterate over the set elements using foreach construct.*

Again, if we want to get the set of `__DPT_x` *employees who are* ***NOT*** `__MGR`*, we can write that query as in line* `S17`*. Line* `S19` *shows another way to get an iterate on composite data structures.*

Now once we have two sets of nodes each with a specific prop-

```
//isDirected
1
//#Vertices #Edges
6  7
//#Vertex property and names
3  __MGR  __DPT_x  __DPT_y
//#Edge property and names
1  __EMAIL
//Vertices with property values
0  1  1  0
1  0  1  0
2  1  0  1
3  0  1  0
4  0  1  0
5  0  0  1
//Edges with property values
0  4  0.4
1  3  1.3
1  5  1.5
2  4  2.4
3  0   3.0
4  1   4.1
5  2   5.2
```

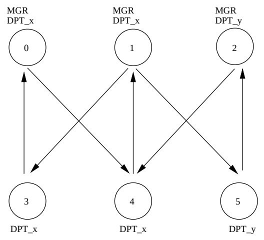

Figure 4: The input file used for graph creation

```
 S1:   define("NUM_VERTICES", "10");
 S2:   define("NOTVISITED", "0");
 S3:   define("VISITED", "1");
       function dfs(v, dfsorder) {
 S4:     if(v.visit == VISITED)
 S5:       return;
 S6:
 S7:     println("vertex visited:  " + v.num);
 S8:     v.visit = VISITED;
 S9:
S10:     dfsorder.pushBack(v);
S11:
S12:     foreach(neighbor; v.getNeighbors())
S13:       dfs(neighbor, dfsorder);
       }

       function main(argv) {
S14:     g = graph();
S15:     g.loadFromFile(argv[0]);
S16:
S17:     dfsorder = array(0);
S18:     dfs(first, dfsorder);
S19:
S20:     for(vertex:  dfsorder ) {
S21:       println(" " + vertex.__id + " " );
S22:     }
       }
```

Figure 5: Example code snippet for dfs traversal on a graph

erty, we can query for the edges between them. For example, lines S28*,* S31 *and* S34 *gives the set of edges emanating from one set of nodes to other set of nodes. For example,* Set1 $\longrightarrow$ Set2 *gives all the directed edges from vertices in set* Set1 *to the vertices in* Set 2*. Similarly,* Set1 $\longleftarrow$ Set2 *gives all the directed edges from vertices in set* Set2 *to the vertices in* Set 1*. Note that operators $\leftarrow$ and $\rightarrow$ are applicable to directed graphs and the operator $\leftrightarrow$ is applicable to both directed and undirected graphs. For directed graphs, it gives all the bidirectional edges between two node sets and for undirected graphs, it returns all the edges between two node sets.* $\square$

EXAMPLE 2. *Figure 5 represents another example implementation the depth first traversal on a graph in our proposed language. In this example we are trying to store the dfs traversal order of the vertices as well. For that we declared an array at line* S17 *and used it to store the traversal order at line* S10*. The method* getNeighbors() *at line* S12 *is again a built-in function and returns the neighboring vertices of the receiver vertex.* $\square$

4. LANGUAGE DESIGN

The syntax of the language is an oversimplified version of C, but without mention of any types. The operations on incompatible types will be error-ed out while interpreting.

We have implemented the tokenizer and syntax analyzer using flex and bison. We are supporting syntax like #include("filename") and #define("PI", "3.14") while doing a single pass of parsing (i.e. Preprocessing of these constructs are done while parsing). This is achieved by using flex's internal stack to manage multiple buffers. The grammar rules are mostly borrowed from [1]. The rules are compiled by bison tool to generate the C parser. We are able to generate the AST corresponding to test-cases confirming to the grammar rules. Our AST is basically a list of function definitions. Each function definition object contains name of the function, a set of formal arguments and a list of body statements. These body statements could be an assignment, loop-statement, function call, etc. The leafs of the AST could be an identifier, int, float, true, false, null, string, vertex, edge or graph.

Some of the key features of the parser is as follows:

- Support of C statements like `if then`, `if then else`, `while`, `for`, `foreach`.

- Support of `break`, `continue` within loop-body and `return` in function-body. As we are representing both loop-body and function-body as compound statements (i.e. anything between "{" & "}"), so we do not have to distinguish these two cases. But we will error-out if break is used inside non-loop body. The detailed semantics of executing a `break`, `continue` and `return` will be discussed in the interpreter runtime section.

- Supporting graph as first class object `graphnode`. The syntax to declare a graph is `g = graph();` which will be represented in AST as an assignment-node with left-node containing an identifier and right node as a function call. Now this function call corresponds to a built in function that returns a `graphnode` (which is of one the leaf nodes of AST) on execution.

- Supporting vertices and edges as first class objects which contains a map to add properties. This feature is useful in various graph algorithms like in dfs traversal 5, we uses a vertex property "visited" to keep track of vertices already explored.

- We are supporting composite data-structures like array, struct and set and iterators on them. Figure 6 represents a code snippet representing some of the operation on these data-structures.

- The language semantics will be same as that of C as we are using a subset of it.'

- All variables are defined as local and are valid only it the scope of the current function (function not block).

- The language specify no constructs for variable declaration and type specification. The interpreter uses some inner data types (like `null, Bool, int, float, string, array, struct, set, graph, vertex, edge`) which can be dynamically changed with assign command.

```
function main(argv) main() {
  arr = array(5);
  i = 0;
  ...
  foreach(var ; arr)
    var = i++;
  ...
  println("-- array items --");
  foreach(var ; arr)
    println(var);
  ...
  st = struct();
  st.number = 42;
  st.pi = 3.14;
  st.str = "bagr";
  ...
  println("-- struct items --");
  foreach(var ; st)
    println(var);
  println("-- struct items using iterator --");
  it = st.iterator();
  while(it.hasNext())
    ‘ println(it.next());
  ...
  g = graph();
  v1 = g.generateVertex();
  v2 = g.generateVertex();
  v3 = g.generateVertex();
  e1 = g.generateEdge(v1, v2);
  e2 = g.generateEdge(v2, v3);
  ...
  v1.color = "red";
  v2.color = "green";
  v3.color = "blue";
  e1.value = 0.5;
  e2.value = 0.4;
  ...
  println("-- vertex set --");
  foreach(var ; g.getVertices())
    println("" + var + ":  " + var.color);
  ...
  println("-- edge set --");
  foreach(var ; g.getEdges())
    println("" + var + ":  " + var.value);
}
```

Figure 6: Example code snippet to show operatons on array, set and struct

5. INTERPRETER RUNTIME

The following are the key features of the runtime:

- The runtime starts with searching for function definition `function main(argv)` and then creates a function call out of it and execute it. While creating the function call it uses the command-line parameters as the actual parameters of the function call.

- The execution of a function call involves finding the corresponding function definition, checking if the number of formal and actuals are equal and then pushing a call stack frame (which contains the mapping between the formal and actual values passed to them) in a global call stack. After that, the function is executed w.r.t the current context(i.e. the top of the call stack).

- The execution of the function involves executing a list of statements. The statements may add further mappings in the current call stack frame. Whenever a name (identifier) is refereed, the mapping in the current context need to be consulted to get the actual value of it.

- The semantics of `break`, `continue` or `return` is supported using the try-catch mechanism of C++.

  For example, while interpreting a `loop-body`, whenever a `break`is encountered, a corresponding `break-object` is thrown, which is caught in a place outside the entire loop execution in order to implement the semantics of break.

  Similarly, while executing a `loop-body`, whenever a `continue` is encountered, a `continue-object` is thrown, which is caught outside the `loop-body` execution so as to skip the current iteration and continue with the `loop-incr` execution (in case of for loop) and `loop-condition-expr` execution.

  And finally, while executing a `function-body`, which is a list of statements, when any one of those statements is a `return`, a `return-object` is thrown, which is caught outside of the loop which is going over that list and in this way the semantics of return is maintained.

- Occurrence of `break` and `continue` within non-loop body triggers an error. While interpreting a node-block (which is a set statements within “{” and “}”), whenever the runtime finds a `break` or `continue` it throws a object. Now if this object is caught inside a non-loop block then error is reported.

- Division by zero and operations on incompatible types are runtime errors.

Also we are supporting a number of built-in functions as shoen in Table 1. The advantage of using is that we can save a lot of interpretation time.

6. EVALUATION

In order to evaluate the performance of our design, we will be coding a number of well known graph related algorithms in our language and run the proposed interpreter on them. The baseline of our evaluation will the the same algorithms implemented in C, compiled by the C compiler and executing the compiled binary.

We are planning to compare the following.

- Runtime performance of the interpreter w.r.t the compiled version of the code.

- Convenience of representation in our language w.r.t C language.

Table 1: List of all the built-in functions provided.

Category	Name of the function	Synopsis
Output	println	Convert the object to a string representation and send the result to the standard output and append newline.
	print	Convert the object to a string representation and send the result to the standard output.
Container	array(size)	Create a new array of specified size.
	struct()	Return a new struct.
	set()	Return a new set
	size(array|struct|set)	Return the size of the argument container.
	union(set, set)	Return union of two sets.
	intersection	Return intersection of two sets.
	difference	Return difference of two sets.
	pushFront(array, elem)	push an element elem to front of array
	pushBack(array, elem)	push an element elem to back of array
	popFront(array)	Remove an elem from front of array
	popBack	Remove an elem from back of array
	front	Get the front elemnt of an array.
	back	Get the back element of an array
Iterator	iterator(object)	Return a copy of a container and set its inner iterator to the beginning.
	hasNext(Object)	Check if a container has a next item.
	next(Object)	Get the next item.
Graph	graph()	Returns a newly created graph
	loadFromFile(graph)	Load a graph from a file
	saveToFile(graph, filename)	Save rhe current state of the graph to a file.
	setDirected(graph, bool)	Set the directed flag of the graph and return the previous value.
	isDirected(graph)	Check if a graph is directed.
	generateVertex(graph)	Create a new vertex in a graph.
	generateEdge(graph, vertex, vertex)	Create a new edge in a graph.
	deleteVertex(graph, vertex)	Delete a vertex from a graph.
	deleteEdge(graph, edge)	Delete an edge from a graph.
	getNumVertices(graph)	Get count of vertices in a graph.
	getNumEdges(graph)	Get count of edges in a graph.
	getVertices(graph)	Get graph’s vertices as a set object.
	getEdges(graph)	Get graph’s edges as a set object.
	getNeighbors(vertex)	Get all neighbors of a vertex as a set object.
	getBeginVertex(edge)	Get the begin vertex of an edge. If the graph is not directed, one of the edge’s vertices will be returned.
	getEndVertex(edge)	Get the end vertex of an edge. If the graph is not directed, one of the edge’s vertices will be returned.
	getAdjacencyMatrix(graph)	Create adjacency matrix from a graph. An array of array will be returned.
	getTransitiveClosure(graph)	Returns the transitive closure of the adjascent matrix representation of the graph. Uses the Floyd Warshall Algorithm.
	getShortestPath(graph, weight, start, end)	Returns an array containing two elements. First, the parent of each vertex in the shortest path tree. Second, the distance of each vertex from start in that shortest path tree. If end == null, the the algorithm will compute the shortest distance of all the vertices from start. Else it stops when the shortest distance of end vertex from start is computed. weight (a string) signifies which of the properties of the edge need to be considered for computing min distance. Uses Dijkstras Algorithm with min heap.
	getMST(graph, weight)	Return the parent of each vertex in the minimum snapping tree. weight (a string) signifies which of the properties of the edge need to be considered for computing min distance. Uses Prim’s Algorithm.
	getVertexSetWithProperty(graph, property, value)	Get the set of vertices with property string equals the value.
	getbfsOrdering(graph)	Get the set of vertices in bfs traversal order.
	getdfsOrdering(graph)	Get the set of vertices in dfs traversal order.

Table 2: Slowdown of GRI w.r.t C implementation.

Algorithm	Fully Scripted time (secs)	C implementation time(secs)	Slowdown
Transitive Closure	1137.40	4.04	281.5
Shortest Path	6.37	0.02	318.5
Minimum Spanning Tree	6.34	0.02	317.0
Graph Coloring	138.17	0.65	212.6

6.1 Runtime Performance Comparison

We have implemented a couple of algorithms in both our language and C. The algorithms are Transitive Closure of a graph using Floyd Warshall's algorithm, Shortest Path using Dijkstra's algorithm, Minimum Spanning Tree using Prim's algorithm and Chaitin's Optimistic graph coloring algorithm. The input graph used in all the cases is a randomly generated graph consisting of vertices = 500, edges = 1000. For Graph Coloring, we used the input graph with vertices = 125, edges = 1000. This is because it takes a lot of time for our interpreter execution to complete. In Table 2, we compared the runtime of our interpreter (henceforth called GRI) with execution time of compiled binary obtained by compiling the C implementation.

Note that, while implementing the algorithms, we used the basic syntax provided by our implementation and **NOT** used any of the built-in functions like `getTransitiveClosure()`, `getShortestPath()` or `getMST()`. We call such implementations as fully scripted.

As we can see that the slowdown are huge. To counter that, we planned to come up with following built-in functions.

- `Graph::getAdjMatrix();`
- `Graph::getTransitiveClosure();`
- `Graph::getShortestPath(wt, start, end);`
  - Returns the shortest distance from vertex start to each vertex in the shortest path tree.
  - Returns parent of each vertex in the shortest path tree.
  - If `end == NULL`, return the shortest path from start to all vertices.
  - If `end != NULL`, return the shortest path from start to end vertex.
  - Weight "wt" (a string) signifies which of the properties of the edge need to be considered for computing min distance.
- `Graph::getMST(wt);`
  - Return the parent of each vertex in the minimum snapping tree.
  - Weight "wt" (a string) signifies which of the properties of the edge need to be considered for computing min distance.

Table 3 shows the speedup of our implementation **with** built-in functions w.r.t **without** using built-in functions (i.e. fully Scripted). The graph used in all the cases consist of vertices = 125, edges = 1000.

Finally, we implemented the above mentioned algorithms using built-in functions and compared with C (Figure 4). The graph used in all the cases consist of vertices = 500, edges = 1000. For Graph Coloring, we used the graph with vertices = 125, edges = 1000.

Table 3: Speedup with built-in functions w.r.t without using built-in functions.

Algorithm	With Built-ins time (secs)	Fully Scripted time(secs)	Speedup
Transitive Closure	0.54	18.19	33.6
Shortest Path	0.08	0.51	6.3
Minimum Spanning Tree	0.05	0.47	9.4

Table 4: Slowdown of our implementation w.r.t C implementation.

Algorithm	GRI Time(secs)	C Time(secs)	Slowdown
Transitive Closure	10.26	4.04	2.5
Shortest Path	0.09	0.02	4.5
Minimum Spanning Tree	0.09	0.02	4.5
Graph Coloring	138.17	0.65	212.6

6.2 Comparison w.r.t Representation Convenience

We will be using the number of lines of code as a metric to compare the convenience that our language is providing w.r.t C.

As we are aware of the fact that this metric is not that useful if we are comparing two languages having different conventions for how much we should put on a line. But as our language is a subset of C, so this metric makes sense, but to yield meaningful results we need to be careful about the following aspects.

- Both the code implementations should produce the same output.
- In both the code implementation, the backward slice of the output instructions should cover the entire code. In other words, there should not be any redundant code which is not contributing to the generated output.
- Both the code implementation should follow similar coding guidelines [3].
- Both the programs need to be equally readable in terms of indentation, newlines, spaces and comments.

We compared the number of lines of static code written in our language with that written in C. Table 5 represents the results of this comparison, which clearly shows that we need fewer lines of code to implement the same algorithm.

Table 5: Comparison of Number of lines of code in our implementation w.r.t C implementation.

Algorithm	Proposed language	C
Transitive Closure (with built-in)	36	129
Transitive Closure (fully scripted)	49	129
Shortest Path (with built-in)	64	191
Shortest Path (fully scripted)	159	191
Minimum Spanning Tree (with built-in)	40	171
Minimum Spanning Tree (fully scripted)	128	171
Graph Coloring	135	213

7. REFERENCES

[1] ANSI C Yacc grammar. `http://www.quut.com/c/ANSI-C-grammar-y-2011.html`.

[2] Graphal: Graph Algorithms Interpreter.

[3] Making The Best Use of C.
`http://www.gnu.org/prep/standards/standards.html#Writing-C`.

[4] E. Adar. GUESS: A Language and Interface for Graph Exploration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’06, pages 791–800, New York, NY, USA, 2006. ACM.