Design a Graph Definition and Querying Language

Report

This language is used to define graphs (directed and undirected) and regular path queries.

1. (45pts) The graph definition language should support:

1.1 (5pts) Defining directed graphs && 1.2 (5pts) Defining undirected graphs

According to the graph definition, this language will meet the graph directed and undirected properties.

Directed Graph: In order to define directed graph "grd" keyword should be used.

Undirected Graph: In order to define undirected graph "gru" keyword should be used.

How to define in general:

Note that:

- Restrictions of name's and value's types are defined above.
- In order to understand well, specific examples is given in tutorial.

Directed: /* Detailed information is in the tutorial under "**Defining Directed Graph**" title */

```
grd myGraph{

    vertex v1<- ("name":value)

    vertex v2<- ("name":value)

    edge v1->v2

}
```

Undirected: /* Detailed information is in the tutorial under "**Undefining Directed Graph**" title */

gru myGraph{

vertex v1 <- ("name": value)

vertex v2 <- ("name": value)

edge v1—v2 <- ("name":value)

}

1.3 (10pts) Defining vertex properties

- A vertex property is some data value associated with a vertex.

- It should be possible to attach multiple properties to a vertex.

- A property should be a (name, value) pair. For instance, if a vertex represents a student, then a vertex property could be ("id", 19945656). Remember that there could be multiple such properties, such as: ("id", 19945656), ("name", "Ali Veli"), ("age", 20).

&&

1.4 (10pts) Defining edge properties

- It should be possible to attach multiple properties to an edge.

Note that: Specific examples are given in the tutorial under the title of "**Defining Vertices" and "Defining Edges**" to understand the definition of them well.

Vertices: /* Detailed information is in the tutorial under "**Defining Vertices**" title */

A vertex can keep multiple properties which consist of ("name":value) pair, where name should be only string and value should be int, float, string, map, list or set.

vertex v1

v1 <- (name:value)

Edges: /* Detailed information is in the tutorial under "**Defining Vertices**" title */

An edge can keep multiple properties which consist of ("name":value) pair, where name should be only string and value should be int, float, string, map, list or set.

vertex v1, v2

edge v1->v2(“name”:value) //with a directed edge

edge v1--v2(“name”:value)//with a undirected edge

1.5 (15pts) A dynamic type system for the property values (property names should be strings)

- Support strings, integers, and floats as primitive types. For instance, in the earlier example, the “id” attribute was using an integer value, whereas the “name” attribute was using a “string” value.

This language is a dynamically typed language, therefore it is not needed to specify any data type for variables. /* Detailed information is in the tutorial under “**Defining Primitive Types: Integer, String, Float**” title */

Examples;

//Integer example

num <- 1 //variable “num” is created and initialized to 1

//float example

num2 <- 1.5 //variable “num2” is created and initialized to floating point 1.5

//String example

str <- “String example” // variable “str” is created and initialized to indicated string, strings should be defined between double-quotes.

- Support lists, sets, and maps as collection types. For instance, it should be possible to have a property like: (“grades”, {“CS315”: 85, “CS101”: 90, “CS666”: 15}). This would be an example of a property, whose value is a map from strings to integers. Arbitrary nesting should be possible as well. For instance, we can have (“grades”, {“CS315”: [85, 80], “CS101”: [90, 98], “CS666”: [15, 3]}). This is an example where the value type is a map from a string to a list of integers.

/* Detailed information is in the tutorial under "**Defining Map, List, Set**" title */

Edge and vertex property values supports sets, maps, and lists as indicated.

Examples;

//Map Example

map is defined as unordered set of ("name":value) pairs

map myMap <- {("string" : "qwerty"), ("key1" : 1), ("key2" : 2), ("key3" : 3) }

//Set Example

Set is defined as unordered collection of unique elements

set mySet <- (1,2,3,4,5,"example")

//List Example

List is defined as ordered collection

list myList <- [1,2,3,4,5]

2. (55pts) The graph querying language should support:

2.1 (30pts) Creating regular path queries. A regular path query is a regular expression specifying a path. A path is a series of edges. Importantly, we are not asking you to evaluate path queries. We are asking you to create a language to express them. A path query should be able to specify:

- (10pts) Concatenation, alternation, and repetition

This language support concatenation, alternation and repetition. " " operator is used for concatenation, "|" operator is used for alternation and "*" and "+" operators are used for repetition.

- With "+" operator, query searches for specific named query to occur one or more times.
- With "*" operator, query searches for specific named query to occur zero or more times.

/* Detailed information is in the tutorial under "**Using "*" and "+" Symbols For Querying, Using Concatenation For Querying and Using "|" Symbols For Querying**" title */

Examples; //how to use

// For concateation " " operator

myGraph § q1 q2 , where myGraph is a graph and q1 and q2 are queries

// For alternation "| " operator

myGraph § q1 | q2 , where myGraph is a graph and q1 and q2 are queries

// For repititon "+" operator

myGraph § q1 + , where myGraph is a graph and q1 is a query

// For repititon "" operator

myGraph § q1 * , where myGraph is a graph and q1 is a query

Using these operations may cause to some edges or vertices may come back to back. There must be an edge between two vertices or there must be a vertex between two edges, so as to perform such query which has only two edges we used ignored edges or ignored vertices. "[ ]" this means the vertex is ignored, "£/£" this means an edge is ignored.

query q1 <- ["name" == "example1"]

query q2 <- ["name" == "example2"]

query q3 <- q1 q2 // This query is converted to q1 £/£ q2

- (10pts) Filters as Boolean expressions, which are composed of predicates defined over edge properties as well as incident vertex properties. For instance, one may want to find all paths of length three (a path of three edges), where the start vertex of the first edge has a vertex property name="CS", the second edge has an edge property code="315", and finally the third edge's end vertex has a vertex property kind="rulez!".

/* Detailed information is in the tutorial under "**Using These Symbols For In Querying Statements**" title in examples */

The user are able to query on a specific graphs in this language.

Example;

myGraph § [] (£ myEdge £ [myVertex])+

where myGraph is a graph, myEdge is an edge and myVertex is a vertex of that query. It is a general expression of that. For more information check the tutorial as indicated above.

- (10pts) Support for existence predicates as well as arithmetic expressions and functions in predicate expressions. For instance: an edge containing or not containing a property with a given name or value; or an edge that has a certain property whose value is greater than a constant threshold; or an edge that has a certain property whose value is a string that starts with "A" (this would require a string indexing function).
/* Detailed information is in the tutorial under "**Using Threshold Type Queries**" title */

Using Threshold Type

This language support ">", "<", ">=" and "<=" operators, where" >" means gretater than," >" means less than ">=" means greater or equal to and "<=" means less or equal to.

General example:

myGraph § ["name" <= value1]

,where myGraph is a graph and "name" is name on the vertex and if value1 is lesser or equal to which is on the vertex it returns true, otherwise returns false.

myGraph § ["name" >= value2]

,where myGraph is a graph and "name" is name on the vertex and if value1 is greater or equal to which is on the vertex it returns true, otherwise returns false.

graph1 § ["name" > value3]

,where myGraph is a graph and "name" is name on the vertex and if value1 is greater than to which is on the vertex it returns true, otherwise returns false.

graph1 § ["name" < value4]

,where myGraph is a graph and "name" is name on the vertex and if value1 is less than to which is on the vertex it returns true, otherwise returns false.

//For specific example check the tutorial as mentioned above.

Using "starts with" Type

This language can check;

- whether a value which corresponds to "name" starts with indicated string. In order to use "starts with", $s is needed to use as a keyword.
- whether a value which corresponds to "name" ends with indicated string. In order to use "ends with", $e is needed to use as a keyword.
- whether a value which corresponds to "name" include a specified string. In order to use "include", $i is needed to use as a keyword.
- whether a substring of value which corresponds to "name" has specified string. In order to use "substring", $d is needed to use as a keyword.

Examples:

//Example of "starts with"

myGraph § ["name" $s "Any String"]

where myGraph is a graph and if the value corresponds the key "name" starts with "Any String", it returns true, otherwise it returns false.

//Example of "ends with"

myGraph § ["name" $e "Any String"]

where myGraph is a graph and if the value corresponds the key "name" ends with "Any String", it returns true, otherwise it returns false.

//Example of "includes"

myGraph § ["name" $i "Any String"]

where myGraph is a graph and if the value corresponds the key "name" includes "Any String", it returns true, otherwise it returns false.

//Example of "distance"

myGraph § ["name" $d(1:3) " Any String"]

where myGraph is a graph and if the substring of the value at given interval, corresponds the key "name" is equal to "Any String" it returns true, otherwise it returns false.

2.2 (15pts) Support having variables in path expressions. For instance, you may want to query all paths of length two where both edges in the path have a property called `name` with the same property value, but the value is not known. In this case, that value becomes a variable.

/* Detailed information is in the tutorial under "**Using Variable Type Queries**" title */

Variables is supported in this language.

a <- " name"

b <- value , where value can be string, int, float, map, set or list.

graph1 § [a==b]

a<- "name"

graph1 § [a== ?b] // If we may want to query for a particular name and a variable value, we must use this query, as indicated assignment.

It is possible to use arithmetic operations inside the query. Example:

a <- "name"

b <- value

c <- value

graph1 § [a== b+c]

2.3 (10pts) Support modularity, that is dividing regular path queries into multiple pieces that are specified separately. This would require giving names to each piece and being able to use those names in a higher-level query.

/* Detailed information is in the tutorial under **"Regular Expression of Query**" title */

In this language, we can define queries such as,

q1 <- ["name"=="example"] £"str1"=="str2"/£

where q1 is a query

graph1 § q1+ // querying for q1 1 or more times.

graph1 § q1* //querying for q1 0 or more times.

graph1 § q1 q2 // concatenation q1 with q2 then querying them.

graph1 § q1 | q2 // querying for q1 or q2

In this way, we support modulartiy as well.