

XAYA 001 Tutorial and Request for Comments

Version: 20060206



Contents

Introduction.....	3
What is XAYA?	3
A Miniature Language (for representing information as networks).....	3
Thinking In Pictures (visualizing querying/navigation/inference in a network).....	5
A Family Tree Network.....	6
An Ecosystem Network.	7
.....	8
A Book Database Network.....	9
.....	9
The XAYA Data Format.....	11
Text File Format.....	11
Python Data Structure Format.....	12
XAYA Interpreter Session.....	15
From Files to Dictionaries: Write, Read and Shelve Operations.....	15
Querying as Navigation: How to Interrogate a Graph.....	18
The Network is the Model.....	25
Utility Functions and Transformations: Rounding Out XAYA.....	28
Functional Programming Style: Look Ma – No Side Effects.....	29
Request for Comments.....	30
Useful References.....	31
XAYA Function Summary.....	32
Input/Out: Handles moving data between various storage formats.....	32
Data Structure transformations:.....	32
Search Operations on Graphs.....	33
Binary Operations on Graphs.....	34
Adding/Dropping Nodes and Edges from a Graph.....	34
XAYA Mythology (the origin of XAYA).....	35
Contact and Participation Information (... help XAYA help you).....	37

Introduction

What is XAYA?

XAYA is for :

- **People who need to work with data but are not programmers.** XAYA provides a small set of commands that allow the user to rapidly model, manage, and transform data to support incorporating business logic into flexible data analysis systems.
- **Software Developers who need to rapidly build and deploy information applications.** XAYA provides a library of core functions that represent information networks and is being expanded to include database functionality, logic, and statistical processing capabilities.

XAYA is a “miniature” language for representing information as networks. XAYA can be used for developing querying and drawing inferences from data/information/knowledge models that are represented as networks. This tutorial is an introduction to XAYA, and a request comments and critique towards making XAYA more useful to help you work and play with information in your business, science, or art.

Dear reader, we now come to a fork in the road.

Fork 1—Okay, make it quick: *If that’s enough of an introduction, proceed directly to “XAYA Interpreter Session” on pg 8 to see XAYA in action. The code is in your friendly neighborhood Python. If you’re not familiar with Python or programming – don’t worry – just keep your eye on the data!*

Fork 2 – The path less taken: *If you want to know more about the concepts guiding XAYA’s development keep on reading below. XAYA’s concepts are meant to be independent of any specific programming language, but the initial implementation is in Python¹.*

A Miniature Language (for representing information as networks)

What is a network? The Oxford dictionary of computing² defines networks several ways, in terms of communication, in terms of electronic circuitry, and mathematically. The mathematical definition is:

“(net) In mathematics, a connected directed graph that contains no cycles. Interconnections involving objects such as telephone, logic gates, or computers could be represented using a connected, but not necessarily directed graph.”

¹ Feel free to port the ideas elsewhere. XAYA is fundamentally an exploration of (not new) ideas about information design, and as such will gain in simplicity by trimming out any ideas specific to a particular programming language

² A Dictionary of Computing. 5th Edn. Oxford University Press. 2004.

XAYA is a language for representing knowledge as networks of related information, for querying that information, for developing models and analyses systems, for finding paths through a maze of related information, and for making deductions. XAYA has features begged, borrowed, stolen from relational databases, logic programming languages such as prolog, and from routing algorithms.

People are good at solving visual puzzles (such as the path through a maze, or driving cross country following a map³) and uses visual abstractions as a guide or map to reality. XAYA uses that visual intuition in data analysis via the central metaphor: **Navigation**.

Navigation represents queries for networks representing a relational database; navigation represents the process of making a logical deduction in a network representing facts defining a problem; navigation represents the route from a high-level knowledge model (e.g. an "Ontology"⁴) through a data-model (e.g. a database schema) to detailed low-level data table (of individual observations). It's networks all the way down!

XAYA applies to that subset of problems that can be represented as a network. Not everything can be represented that way – but a large number of things can

.... Database-like manipulations, logical deductions, are all translated into finding routes through a network of related information. XAYA's design constraints are to balance smallness with simplicity, with flexibility.

XAYA 's core language is meant to be **small**. Additional functionality should be built by combining parts of the core language in new combinations (combinatorial explosion is your friend). To ensure portability across the various flavours of Python, XAYA's core implementation is limited to using only those modules that come with the Python Standard Library. As XAYA develops, I hope to keep it **simple** enough that a domain expert (in business, in science, or in art/design) who is say familiar with spreadsheets or canned statistics software, can use it to build and explore interesting models in their field without a great investment in time/effort to get started, but that it provides a **flexible** toolkit for a developer to build data driven modeling applications. Hopefully it provides a bridging language between the world of an application domain and the world of the software developer. The domain expert focuses on representing their knowledge diagrammatically as a graph (imagine sketching a design as calligraphy). The software developer works on converting these pictures into application code.

Simplicity is achieved by having a uniform data structure – the network representation (= atomic "nouns" in the mini-language). Networks are the input and output of most XAYA functions. Flexibility is via developing complex models out of networks, being able to represent a wide range of data structures (text files, databases, grid based imagery) as a network, and by composing functions into series of data transformations. Operations on networks (= "verbs" in the mini-language), create new networks.

³ I have to take this on faith, as I usually get lost in parking lots, with or without a map.

⁴ An ontology describes concepts and their relationships. It's a very old philosophical idea finding new lanterns in the internet under the waxing vision of "the semantic web".

Version 001 of XAYA is necessarily sparse – it is meant to convey the basic ideas, and solicit feedback for the next iteration. We begin with some pictures of different things that can be represented as networks: a family tree, a book collection database, and food relationships in an ecosystem.

Thinking In Pictures (visualizing querying/navigation/inference in a network)

XAYA currently has only a command line interface – so where does visualization fit in? Visualization is central to certain styles of problem solving. XAYA forces one to represent problems as a network. Thus, we can visualize any problem as sort of a “map” of directed arrows connecting the objects/ideas we wish to represent. This map is our working model of the problem or the facts at hand, as we understand it.

Below are several pictures of networks. In each picture there are nodes (represented either as labeled boxes or “thought” clouds), which represent objects. The nodes are connected by arrows. Each arrow has a head and a tail end. The object at the tail end of an arrow is the Parent Node. The object at the arrowhead end is the Child Node. The arrows represent directed relationships amongst objects. That's it.

Here is a very simple version of a network with 7 nodes. Of the 7 nodes, two are “root nodes (nodes with no parents), three terminal nodes (nodes with no children) two interior nodes (nodes with parents and children). One of the terminal nodes, has two parents (the two root nodes). In this particular case, the “Nodes” represent geometric objects such as might appear in an imagery or geographic information data set, and the arrowhead relation represents the relationship “contains”. I.e. the parent nodes contain the child nodes.

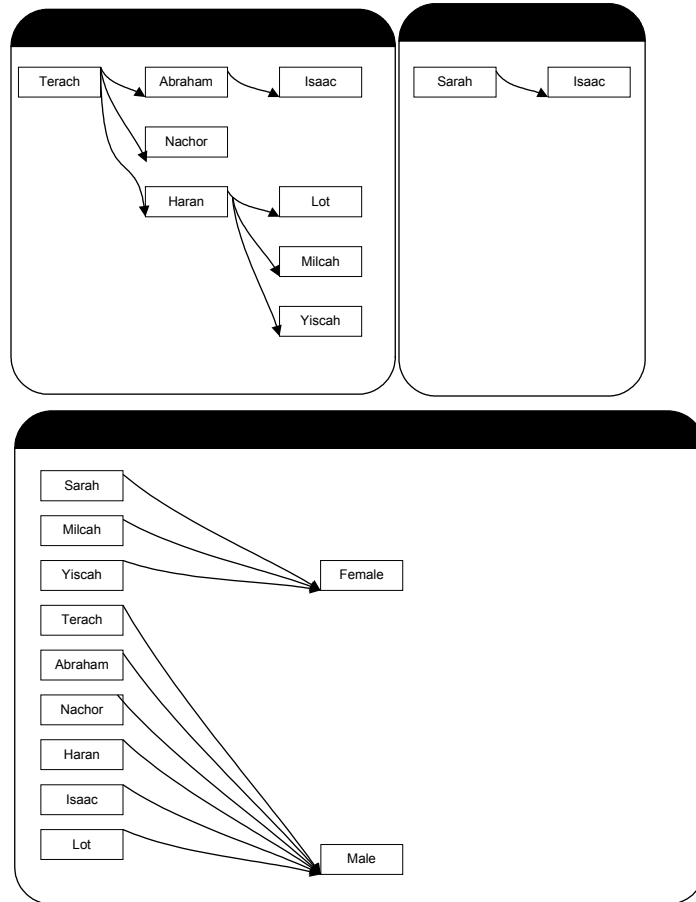
Put in a Picture – ParentNode – and Two Child Nodes.

While the pictures that follow simple enough we can solve any problem simply by staring at the network picture long enough – these ideas naturally scale to networks you can't grok in a single glance.

Define parts of a graph: roots, leaves, terminal leaves.

A Family Tree Network⁵

Here is a picture of a family tree.



In this picture the nodes (labeled boxes) represent people. The arrows represent relationships. There are several networks. One network represents father→child relations.

⁵ Taken from pg. 12 of "The Art of Prolog" by Sterling and Shapiro. MIT Press. 1994.

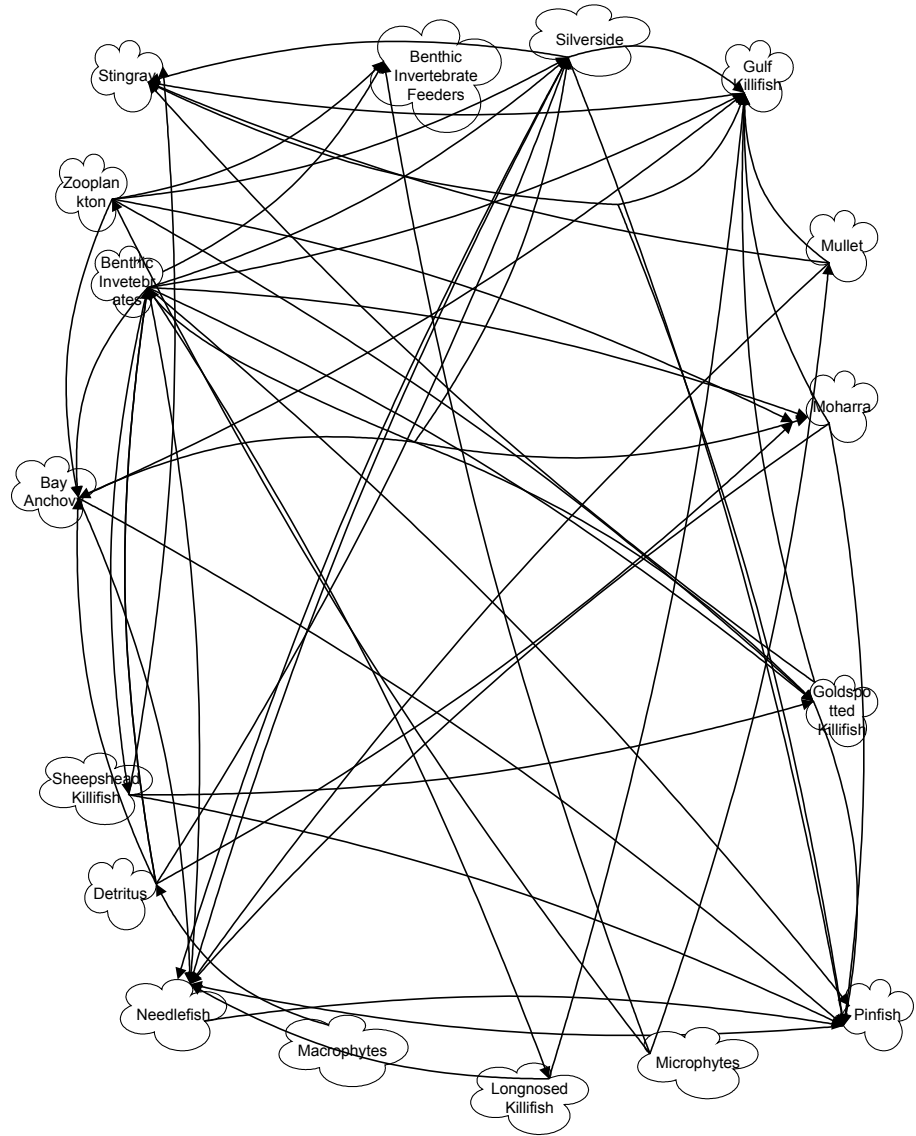
Another network represents mother→child relations. Finally two disconnected networks⁶ represent person→gender relations. The people in this example are characters from the Old Testament, and their genealogy seems to be the equivalent of “hello world” in Logic Programming (somewhere there must be a post-modernist penning the manifesto: “Religious Iconography in AI”).

Given these network pictures, one can ask questions such as: Who are Isaac’s parents? One can define new relations such as “Grandparents” and represent them as networks.

An Ecosystem Network⁷.

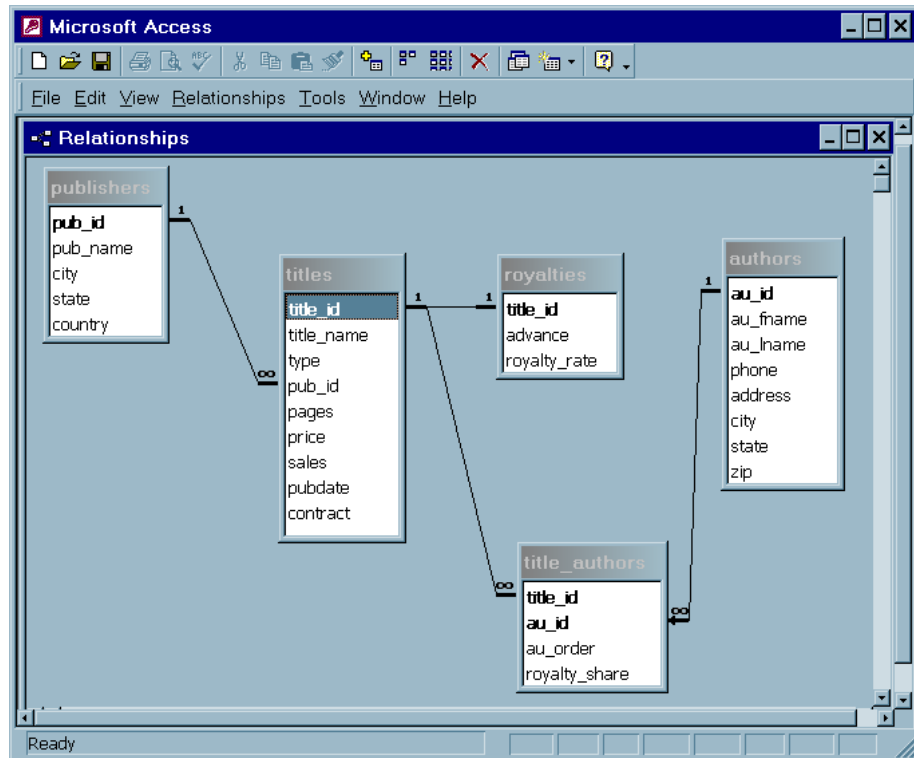
⁶ Disconnected, as in there are no nodes in common between the two networks. If this had been a network representing people in T.S. Eliot’s “The Wasteland” – the character Tiresias would have an arrow to both genders.

⁷ Simplified from from Figure 7.1 on pg. 122 of “Ecology the Ascendent Perspective” by R.E. Ulanowicz. My apologies to the author, if I missed any arrows!



Admittedly this portrait of network is no work of art. But it represents the kind of complexity ecologists must deal with. The network represent a coastal salt marsh. The nodes (clouds) are species/components of the marsh. The arrows can be interpreted as the relationship “eaten by” as in “Zooplankton are eaten by Bay Anchovy”. At first glance, it looks quite chaotic. But is there some order within this seeming chaos?

A Book Database Network⁸



This network looks rather different from the previous pictures, as it's in a standard database representation. In this case, instead of “arrows” connecting items in the model, there are lines,

⁸ Based on the “Books” database that accompanies “SQL Visual Quickstart Guide” by Chris Fehily which is available online at: www.peachpit.com/vqs/sql

with a “1” side, and with a “∞” side (the relationship implying “1-to-many”). The “1” side is like the tail of our arrow, and the “∞” side is like the head of our arrow in the previous networks. This kind of network is often called an “entity relationship diagram” or “ERD”.

The way to interpret it, is as a set of statements about “entities” (which are like the nodes of our network). The statements reflected in this database are things like: “A publisher has many titles. A title can have many authors. An author can have written many titles.

Two “entities”, named “titles” and “royalties” have a “1” on both sides of the relationship? How do we interpret this in our network model. In this particular case, the implied relationship is “Every title has a royalty associated with it”. Thus, we can represent it as having a being a directed arrow from a “title” node to a “royalty” node.

Now, each of these “entities” are represented by data tables. The data tables have columns, and they have rows. We can model all this in XAYA – but now, rather than having a single network, what we end up with is several networks that represent a hierarchy of information from low level details of data, to relationships amongst tables.

The particular networks we would need to represent in XAYA are:

- 1.The entity relation network where nodes are entities, and arrows represent 1-to-many or 1-to-1 relationships amongst entities.
- 2.A network representing tables and columns, where the parent nodes are table names, and the child nodes are column names.
- 3.A network representing “primary keys” where the parent nodes are the table names, and the child nodes are the keys. This network is essentially a subset of the previous network. A primary key represents a column (or several) that uniquely identify a row of data.
- 4.A network for each data table where for each row of data, there is a parent node with arrows to each of the column values for that row.

This set of networks is represented in the figures below. With this structure – a network of networks -- , we can simulate some of the operations of a relational database directly within XAYA. We can also use XAYA to build a small program that translates between a query defined in XAYA and one defined in SQL, the native language for relational databases. We can call this program a “query engine”, it will provide us with an extended example of how XAYA can be used to work with relational databases.

Thus, in our three examples – geneology, ecology, databases – we get an idea of how XAYA can be used to ask questions about a wide range of networks, and an idea of how useful it is to think in pictures of networks to model problems in various areas.

The XAYA Data Format

Text File Format

The basic text file format for XAYA is as follows:

ParentNode: ChildNode1, ChildNode2, ChildNode3 ...

Each Parent Node and its Child Nodes are represented on a single line (that terminates with an invisible newline character when you hit "Enter" or "Return" on your keyboard)

The Parent Node is followed by a ":".

The ":" is followed by a list of the Child Nodes.

The text file "biblicalDad.txt" for the Relation "father" in the previous section would look like this:

Terach:Abraham, Nachor, Haran

Nachor:

Haran:Lot, Milcah, Yiscah

Abraham:Isaac

Line 1: Terach (ParentNode) is the father of Abraham (ChildNode1), Nachor (ChildNode2) and Haran (ChildNode3).

Line 2: Nachor (ParentNode) has no children.

Line 3: Abraham (ParentNode) is the father of Isaac (ChildNode).

Python Data Structure Format

In Python, the text-file format is translated into a data structure called a dictionary. In our case, we represent a network as a dictionary of Lists⁹

A dictionary consists of keys and values. In our dictionaries, keys are used to represent a ParentNode in the network, and the values are represented as lists of ChildNodes. In the Python interpreter, the structure does not look very different from the text file, except we have to use some extra symbols to define containers.

A dictionary (i.e. a network) is contained between the “{“ and “}” symbols.

A list (i.e. all the ChildNodes of a ParentNode) is contained between the “[“ and “]” symbols.

Each item in the dictionary is a Key : Value pair, separated by a comma.

The values are always lists in XAYA.

Each member in the list, is separated by commas. And an empty list, is a list with no members.

The basic dictionary format is¹⁰:

```
{ParentNode1: ChildNode11, ChildNode12, ChildNode13 ...}
```

The dictionary corresponding to the textfile looks like this in the python interpreter:

```
>>> biblicaldad = {'Terach' : ['Abraham', 'Nachor', 'Haran'],  
... 'Nachor' : [ ],  
... 'Abraham' : ['Isaac'],  
... 'Haran': ['Lot', 'Milcah', 'Yiscah']}  
>>>
```

The interpretation of each Key : Value pair (i.e. the key, and it's associated list) is as for the text file. Note that Nachor, who has no sons, is associated with an empty list container.

In this tutorial we'll follow these conventions. Formats, or generalized patterns for data or programming are in bold italics. Interpreter sessions are in a Courier font. The items we would type in, are in Bold.

Now, if we enter the name of the dictionary in the interpreter, we will see it's contents:

⁹ The network representation is from “Python Patterns – Implementing Graphs” by GvR, available at: <http://www.python.org/doc/essays/graphs.html> . This is perhaps the simplest network representation. As XAYA functionality is increased, it might have to be expanded to – see Request for Comments section.

¹⁰ In the examples below – a format template pattern is in bold italics. Examples of an interpreter session are in “courier” font, with the parts we are emphasizing in bold.

```
>>> biblicaldad
{'Terach': ['Abraham', 'Nachor', 'Haran'], 'Nachor': [], 'Haran':
 ['Lot', 'Milcah', 'Yiscah'], 'Abraham': ['Isaac']}
>>>
```

We can access a particular item in a dictionary with the following syntax:

DictionaryName[ParentNode]

As in:

```
>>> biblicaldad['Terach']
['Abraham', 'Nachor', 'Haran']
>>>
```

We can access a particular member of a list with the following syntax:

DictionaryName[ParentNode][list index number]

As in:

```
>>> biblicaldad['Terach'][0]
'Abraham'
```

Which gives us the first listed son of Terach (indexes begin with 0, rather than 1).

If we wanted the third son of Terach, we enter the following:

```
>>> biblicaldad['Terach'][2]
'Haran'
>>>
```

And if we wanted simply to check the first three letters in the name of the third son of Terach we simply type in:

```
>>> biblicaldad['Terach'][2][0:3]
'Har'
>>>
```

If we type in a particularly long line in the interpreter, it will carry on the line. However the formatting of this tutorial will cause the line to break. For example:

```
>>> coastalsaltmarsh = {'Microphytes' : ['Zooplankton', 'Mullet',
'Benthic Invertebrate Feeders'],
... 'Macrophytes' : [ 'Detritus'],
```

represents two lines in the interpreter in the interpreter session, though it takes up three lines when in the formatting of this document. The first line is at ">>>". Our continuation on the next line is represented by "...".

So – if you see something in bold (text to enter) covering two lines, but no "...", you should enter it as a single line in the interpreter.

This should provide all the information you need to create a file in XAYA's network format, or directly enter a network into the interpreter, and to query the contents of such a network.

XAYA Interpreter Session

In this section, we use Python. If you're not familiar with Python or programming, don't worry – we're going to use very few programming constructs in these examples¹¹. If you don't have Python installed, download it from www.python.org and go through an introductory tutorial to familiarize yourself with the interpreter. To get started with this tutorial, place the file "Xayacore_001.py" somewhere on the Python Path¹² (a location the interpreter searches for relevant scripts) – for example something like: "C:\Python23\Lib\site-packages". Locations on the Python Path will vary from system to system, and depend on your installation.

From Files to Dictionaries: Write, Read and Shelve Operations

Import the Xaya module:

```
>>> import Xayacore_001
```

We're now ready to begin.

```
>>> # Lets enter a pair of small graphs
>>> hybrid_candy1 = {'Chocolate' : ['Peanut Butter Cup'],
... 'Peanut Butter' : ['Peanut Butter Cup'] }
>>> hybrid_candy2 = {'Chocolate' : ['Caramilk Bar'],
... 'Caramel' : ['Caramilk Bar'] }
>>>
```

Now let us write these graphs out to a text file. Python has a convenient help function that allows you to access any documentation the programmer included.

```
>>> help(Xayacore_001.writeGraph)
```

Help on function writeGraph in module Xayacore_001:

```
writeGraph(graph={}, filepath='defaultFilePath')
    Inverse of readGraph. Stores a XAYA format dictGraph as a text
file
```

¹¹ . "Python Visual Quickstart Guide" by Chris Fehily covers Python's "core" feature set succinctly, with a strong emphasis on interpreter sessions. The "book" database used in the examples below is drawn from "SQL Visual Quickstart Guide", also by Chris Fehily – and that text provides a point of comparison between "XAYA style queries" and a more traditional database style. An excellent online introduction to Python is available at: <http://diveintopython.org/> . "Dive Into Python" will take you from very basic Python concepts to more advanced features – at a fairly challenging (and rewarding) pace. There are many other books and online tutorials available on Python – just google to find out which ones suit you.

¹² The Python Path are those locations on the Python Interpreters search path – where it looks to find modules.

Reading the help we see that we have to supply the writeGraph function with two objects – a graph, and a filepath. In the absence of a user entry, the function assumes an empty graph “{}” and a file named simply ‘defaultFilePath’. All functions in XAYA contain some basis help documentation, and the more complex functions include a summary of the algorithms behind the function.

```
>>> Xayacore_001.writeGraph(graph = hybrid_candy1, filepath =
'/XAYA/devcore/hybrid_candy1_X001.txt')
<open file '/XAYA/devcore/hybrid_candy1_X001.txt', mode 'a+' at
0x01674A20>
>>> Xayacore_001.writeGraph(graph = hybrid_candy2, filepath =
'/XAYA/devcore/hybrid_candy2_X001.txt')
<open file '/XAYA/devcore/hybrid_candy2_X001.txt', mode 'a+' at
0x01674B20>
```

>>>
This creates two files in the ‘hybrid_candy1_X001.txt’ and ‘hybrid_candy2_X001.txt’ in the file location “C:\XAYA\devcore”.

The first file contains this text:
Peanut Butter:Peanut Butter Cup
Chocolate:Peanut Butter Cup

And you can guess what the second file contains. Mmm, good.

Having taken a graph, and written it to a file, we can go in the opposite direction – taking the file and writing it to a graph, and test that this new graph (from the text file) is equivalent to the original graph we entered into the interpreter by hand. .

```
>>> hybrid_candy1_redux = Xayacore_001.readGraph(filepath =
'/XAYA/devcore/hybrid_candy1_X001.txt')
```

```
-----
Opening file /XAYA/devcore/hybrid_candy1_X001.txt
>>> hybrid_candy1_redux == hybrid_candy1
True
>>>
```

In addition to writing a graph to a file, we can “shelve” it. When we type a graph into the Python interpreter, using the dictionary datastructure, the graph exists in memory only. If we shut down the interpreter, without writing the graph to a file, it is lost. A “shelf” is a persistent (i.e. on disk) dictionary-like object. For most purposes, we can treat it just like a dictionary in the interpreter. So, we can “shelve” a graph in one session, and re-use it in the next session.

```
>>> shelfHybridCandy1 = Xayacore_001.shelveGraph(graph =
hybrid_candy1, filepath = '/XAYA/devcore/shelfHybridCandy1')
>>> shelfHybridCandy1
{'Peanut Butter': ['Peanut Butter Cup'], 'Chocolate': ['Peanut
Butter Cup']}
```

```

>>> shelfHybridCandy1 == hybrid_candy1
True
>>> shelfHybridCandy1.close()
>>> import shelve
>>> openshelve = shelve.open(filename =
'/XAYA/devcore/shelfHybridCandy1')
>>> openshelve
{'Peanut Butter': ['Peanut Butter Cup'], 'Chocolate': ['Peanut
Butter Cup']}
>>> openshelve == hybrid_candy1
True

```

Note that in this case, we first created a shelf object “shelfHybridCandy1”, tested it was equivalent to the original “hybrid_candy1”, then closed it. To reopen the shelf object “shelfHybridCandy1” we first had to import the shelve module into the interpreter, then used it’s “shelve.open” function to assign the data to the “openshelve” variable. Finally we again tested that “openshelve” and “hybrid_candy” have the same values.

We’ve been doing things step-by-step so far. But as previously mentioned, the outputs from one XAYA function can be the inputs for the next XAYA functions – so we can compose functions together into a sequence such as:

```

>>> read_then_shelveCandies = Xayacore_001.shelveGraph(graph =
Xayacore_001.readGraph(filepath =
'/XAYA/devcore/hybrid_candy2_X001.txt'), filepath =
'/XAYA/devcore/shelfHybridCandy2')
-----

```

```

Opening file /XAYA/devcore/hybrid_candy2_X001.txt

```

```

>>> read_then_shelveCandies == hybrid_candy2
True
>>>

```

In this case, we’ve read a text file, then moved it into a shelve object in a single step and assigned the results to the variable “read_then_shelveCandies”. Finally, we tested whether “read_then_shelveCandies” has the same values as “hybrid_candy2”.

As we’ll see later in this tutorial, composing functions in this way, allows one to accomplish quite a bit by composing a small set of “base functions” in different sequences. Those who are used to statistical data analysis, accomplish the same thing by imposing a sequence of data transformations.

Okay, we’re now ready for some more complex, if less tasty graphs.

Querying as Navigation: How to Interrogate a Graph

We now turn to the key idea in XAYA – that querying a graph is an exercise in navigation. XAYA incorporates a number of simple functions by which we can interrogate a graph. It's easiest to explain this concept by going through a few examples demonstrating increasingly complex questions one may ask of a graph.

Perhaps the simplest question we can ask of a graph are “What are its Roots and Leaves”.

Again, to find out about a XAYA function, we can invoke the help function.

```
>>> help(Xayacore_001.findRootsLeaves)
Help on function findRootsLeaves in module Xayacore_001:

findRootsLeaves(graph={})
    Given a Graph, finds Root and Leaf Nodes for the Graph
    Preconditions: Graph in Xaya Format
    Postconditions: Returns a Graph with two keys: "Roots" and
    "Leaves", each with their valuelist.
    Invariants:

    Usage:

    Definition:
        # A "keyset" is the set of Keys in a Graph.
        # A "valueset" is the set of unique values in a Graph
        # A "Root" is a member of keyset that is not a member of
valueset = 'rootset'
        # A "Leaf" is a member of valueset that is not a member of
keyset = 'leafset'
```

Now lets ask some questions about genealogy¹³ and about ecology.

Who is the root father?

```
>>> biblicaldad = Xayacore_001.readGraph(filepath =
'/XAYA/devcore/BiblicalDad_X001.txt')
-----

Opening file /XAYA/devcore/BiblicalDad_X001.txt
>>> biblicaldad
{'Terach': ['Abraham', 'Nachor', 'Haran'], 'Nachor': [''],
'Haran': ['Lot', 'Milcah', 'Yiscah'], 'Abraham': ['Isaac']}
>>> root_biblicaldad = Xayacore_001.findRootsLeaves(biblicaldad)
```

¹³ Look at “The Art of Prolog” by Sterling and Shapiro, chapter 1, for examples of how similar queries are represented in a more conventional logic programming style.

```
>>> root_biblicaldad
{'Leaves': ['', 'Yiscah', 'Isaac', 'Lot', 'Milcah'], 'Roots':
 ['Terach']}
>>>
```

Who are Lot's male ancestors?

```
>>> ancestors_of_Lot = Xayacore_001.findPathsUp(graph =
biblicaldad, endnode = 'Lot')
>>> ancestors_of_Lot
{'Terach': [['Terach', 'Haran', 'Lot']], 'Haran': [['Haran',
'Lot']]}
```

The “findPathsUp” function returns a graph whose nodes are nodes that are “upstream” from the selected node, and whose value list, is the path to the selected node.

Terach and Haran are ancestors of Lot, because there is a path from them to Lot.

Who is Lot's grandfather?

```
>>> grandfather_of_Lot = Xayacore_001.findPathsUp(graph =
biblicaldad, endnode = 'Lot', operator = '==', pathlength = 3)
>>> grandfather_of_Lot
{'Terach': [['Terach', 'Haran', 'Lot']]}
```

Lot's grandfather is Terach. We define a grandfather as a path upstream from Lot that has length of three (Grandfather, father, child).

Who are Terach's grandchildren?

```
>>> grandchildren_of_Terach = Xayacore_001.findPathsDown(graph =
biblicaldad, startnode = 'Terach', operator = '==', pathlength =
3)
>>> grandchildren_of_Terach
{'': [['Terach', 'Nachor', '']], 'Milcah': [['Terach', 'Haran',
'Milcah']], 'Isaac': [['Terach', 'Abraham', 'Isaac']], 'Lot':
[['Terach', 'Haran', 'Lot']], 'Yiscah': [['Terach', 'Haran',
'Yiscah']]}
```

We define grandchildren as those paths downstream from a chosen ancestor which have a pathlength of 3. The ‘==’ operator implies we want only those paths that have exactly three items.

This result is a little hard to read. So we'll use another very helpful Python function to improve the format, called 'pprint' for "pretty print".

```
>>> import pprint
>>> pprint.pprint(grandchildren_of_Terach)
{'': [['Terach', 'Nachor', '']],
 'Isaac': [['Terach', 'Abraham', 'Isaac']],
 'Lot': [['Terach', 'Haran', 'Lot']],
 'Milcah': [['Terach', 'Haran', 'Milcah']],
 'Yiscah': [['Terach', 'Haran', 'Yiscah']]}
>>>
```

The grandchildren of Terach are Isaac, Lot, Milcah, Yiscah. The first element, '' – an empty quote is due to our format convention of assigning an empty list, in the case where there are no children. Knowing that, we could refine our definition a bit, using a bit of additional code:

```
>>> for key in grandchildren_of_Terach:
...     if key <> ' ':
...         print key
...
Milcah
Isaac
Lot
Yiscah
>>>
```

Let us now turn to Ecology and ask some questions about who's eating what. Lets begin by representing a coastal saltmarsh ecosystem in our graph format, entering the data into the interpreter¹⁴.

```
>>> coastalsaltmarsh = {'Microphytes' : ['Zooplankton', 'Mullet',
'Benthic Invertebrate Feeders'],
... 'Macrophytes' : [ 'Detritus'],
... 'Detritus' : ['Benthic Invertebrates', 'Moharra',
'Silverside', 'Benthic Invertebrate Feeders', 'Bay Anchovy'],
... 'Zooplankton' : ['Bay Anchovy', 'Benthic Invertebrate
Feeders', 'Silverside', 'Moharra', 'Goldspotted Killifish'],
... 'Benthic Invertebrates' : ['Longnosed Killifish', 'Sheepshead
Killifish', 'Goldspotted Killifish', 'Pinfish', 'Needlefish',
... 'Moharra', 'Gulf Killifish', 'Silverside', 'Benthic
Invertebrate Feeders', 'Bay Anchovy'],
... 'Stingray' : [''],
... 'Bay Anchovy' : ['Pinfish', 'Needlefish', 'Gulf Killifish'],
... 'Needlefish' : [ 'Pinfish'],
... 'Sheepshead Killifish' : ['Pinfish', 'Gulf Killifish',
'Stingray' ],
```

¹⁴ The data is from Figure 7.1 on pg. 122 of "Ecology the Ascendent Perspective" by R.E. Ulanowicz. It represents the paths of carbon transfers among 17 major species of a coastal salt marsh in Florida. "transferring carbon" is a polite way of saying "getting eaten".

```

... 'Goldspotted Killifish' : ['Pinfish', 'Gulf Killifish',
'Stingray' ],
... 'Gulf Killifish' : ['Stingray' ],
... 'Longnosed Killifish' : ['Needlefish', 'Gulf Killifish'],
... 'Silverside' : ['Pinfish', 'Needlefish', 'Gulf Killifish',
'Stingray'],
... 'Moharra' : ['Pinfish', 'Needlefish', 'Gulf Killifish'],
... 'Benthic Invertebrate Feeders' : [''],
... 'Pinfish' : [ 'Needlefish'],
... 'Mullet' : ['Stingray', 'Gulf Killifish', 'Needlefish']}

```

Note – for those species who are not eaten – we’ve entered [''] as the ChildNode – signifying an empty list, with an empty string as its only entry.

Who eats not others? Who is not eaten?

```

>>> Xayacore_001.findRootsLeaves(graph=coastalsaltmarsh)
{'Leaves': [], 'Roots': ['Microphytes', 'Macrophytes']}

```

Microphytes and Macrophytes “root” this community by being the original source of food (being “micro” and “macro” sized aquatic plants).

As “Leaves” we find the symbol []. With a bit of code, we can identify which species it refers to.

```

>>> for node in coastalsaltmarsh:
...     if coastalsaltmarsh[node] == [ ' ' ] :
...         print node
...
Benthic Invertebrate Feeders
Stingray

```

There only species which “eats but is not eaten” in the coastal salt marsh appear to be Benthic Invertebrate Feeders and Stingrays. They’re the top of the heap.

Are there any “food chains” of more than 5 links beginning with Macrophytes?

```

>>> greaterthan5 = Xayacore_001.findPathsDown(coastalsaltmarsh,
startnode='Macrophytes', operator = '>', pathlength=5)
>>> greaterthan5
{'Pinfish': [['Macrophytes', 'Detritus', 'Benthic Invertebrates',
'Longnosed Killifish', 'Needlefish', 'Pinfish'], ['Macrophytes',
'Detritus', 'Benthic Invertebrates', 'Moharra', 'Needlefish',
'Pinfish'], ['Macrophytes', 'Detritus', 'Benthic Invertebrates',
'Silverside', 'Needlefish', 'Pinfish'], ['Macrophytes',
'Detritus', 'Benthic Invertebrates', 'Bay Anchovy', 'Needlefish',
'Pinfish']], 'Needlefish': [['Macrophytes', 'Detritus', 'Benthic

```

```
Invertebrates', 'Sheepshead Killifish', 'Pinfish', 'Needlefish'],
['Macrophytes', 'Detritus', 'Benthic Invertebrates', 'Goldspotted
Killifish', 'Pinfish', 'Needlefish'], ['Macrophytes', 'Detritus',
'Benthic Invertebrates', 'Moharra', 'Pinfish', 'Needlefish'],
['Macrophytes', 'Detritus', 'Benthic Invertebrates', 'Silverside',
'Pinfish', 'Needlefish'], ['Macrophytes', 'Detritus', 'Benthic
Invertebrates', 'Bay Anchovy', 'Pinfish', 'Needlefish']]}
```

Apparently, there are some rather long paths, which could be a concern in certain cases – for example if toxins were known to concentrate in the species at the end of long food-chain paths.

Now, lets build on that insight, and try and begin comparing paths for two groups of fish. I've rather arbitrarily chosen four species, and put them into two groups “ScaryFish” and “MildFish”

```
>>> marshquery = { 'ScaryFish' : ['Stingray', 'Needlefish'],
'MildFish' : ['Mullet', 'Bay Anchovy'] }
```

```
>>> pathstoplants =
Xayacore_001.findUniquePathsToRoot(graph=coastalsaltmarsh,
selections=marshquery)
```

The resulting graph is quite long so we will use the pretty print function to make the results easier to read. Since the results go on for a while, we'll truncate partway

```
>>> pprint.pprint(pathstoplants)
{'MildFish/Macrophytes': [['Macrophytes',
'Detritus',
'Benthic Invertebrates',
'Bay Anchovy'],
['Macrophytes', 'Detritus', 'Bay Anchovy']],
'MildFish/Microphytes': [['Microphytes', 'Mullet'],
['Microphytes', 'Zooplankton', 'Bay Anchovy']],
'ScaryFish/Macrophytes': [['Macrophytes',
'Detritus',
'Benthic Invertebrates',
'Sheepshead Killifish',
'Stingray'],
['Macrophytes',
'Detritus',
'Benthic Invertebrates',
'Goldspotted Killifish',
'Stingray'],
['Macrophytes',
```

... results truncated because they go on for a long time, and are mainly of interest to coastal marsh ecologists ...

The results are a graph where the ParentNodes are of the format "

SelectedGroup/Root

Such as MildFish/Macrophytes,

And the ChildNodes are no longer individual items, but themselves paths – so what is being listed are alternative paths to the roots (Macrophytes and Microphytes).

If you looked at the full graph, it would be clear that the so-called MildFish have fewer alternate paths to the root of this ecosystem (i.e. the aquatic plants: macrophytes and microphytes).

Finally, lets find all the paths that exist by which Benthic Invertebrates can end up (eventually) in Stingrays

```
>>> BenthToSting =
Kayacore_001.findAllPathsAsLists(graph=coastalsaltmarsh,
startnode='Benthic Invertebrates', endnode='Stingray')
>>> pprint.pprint(BenthToSting)
[['Benthic Invertebrates', 'Sheepshead Killifish', 'Stingray'],
 ['Benthic Invertebrates', 'Goldspotted Killifish', 'Stingray'],
 ['Benthic Invertebrates', 'Silverside', 'Stingray']]
>>>
```

There are three different “intermediary species” who eat Benthic Invertebrates before they themselves are eaten by Stingrays.

Let us do one last query around the interesting idea of ecosystem stability or resilience. We have 17 species in this ecosystem. If those 17 species were arranged on a single path – you could imagine the system would be very unstable – loss of a single species would bring the system crashing down. So – we have the idea that at a very simplified level (for an ecologist), having numerous paths will contribute to the stability of an ecosystem, and the ability of the ecosystem as a whole to withstand a perturbation in a particular component species.

How many unique paths are there in this ecosystem?

```
>>> speciespathsquery = { 'species' : ['Pinfish', 'Bay Anchovy',
'Macrophytes', 'Needlefish',
... 'Benthic Invertebrate Feeders', 'Zooplankton', 'Microphytes',
'Goldspotted Killifish',
... 'Longnosed Killifish', 'Benthic Invertebrates', 'Moharra',
'Silverside', 'Gulf Killifish',
... 'Stingray', 'Mullet', 'Detritus', 'Sheepshead Killifish'] }
```

```

>>> ascendencypaths =
Xayacore_001.findUniquePathsToRoot(graph=coastalsaltmarsh,
selections=speciespathsquery)
>>> ascendencypaths.keys()
['species/Microphytes', 'species/Macrophytes']
>>> len(ascendencypaths['species/Microphytes'])
18
>>> len(ascendencypaths['species/Macrophytes'])
33
>>>

```

In this assemblage of 17 species, there are 51 unique paths. Interestingly, there are twice as many paths rooted in Macrophytes as rooted in Microphytes.

What is the distribution of these paths?

```

>>> macrophyte_path_lengths = []
>>> microphyte_path_lengths = []
>>> for path in ascendencypaths['species/Microphytes']:
...     microphyte_path_lengths.append(len(path))
...
>>> for pathlength in range(1, max(microphyte_path_lengths) + 1):
...     print pathlength, microphyte_path_lengths.count(pathlength)
...
1 0
2 1
3 2
4 4
5 11
>>> for path in ascendencypaths['species/Macrophytes']:
...     macrophyte_path_lengths.append(len(path))
...
>>> for pathlength in range(1, max(macrophyte_path_lengths) + 1):
...     print pathlength, macrophyte_path_lengths.count(pathlength)
...
1 0
2 0
3 1
4 2
5 15
6 15
>>>

```

If you're new to Python , or programming, don't worry about what I just did, worry about what it means. I created histograms (which I've highlighted in yellow) of lengths for the unique paths in a few lines of code. For Microphytes – there are 11 paths of length 5, with a rapid fall down in number of paths below length 5. For Macrophytes there are 15 paths

of length 6, and 15 paths of length 5, followed by a rapid fall down in number of unique paths of lower length.

Could the tendency towards longish unique paths, rather than shortish unique paths be an indicator of the connectedness of an ecosystem, the integration of its components? Would you expect a characteristic distribution of path lengths for oligotrophic (nutrient poor) ecosystems, versus eutrophic (nutrient rich) ecosystems?

XAYA's graph representation currently does not allow one to "carry baggage on an arc" – that is, we are representing in our representation the fact that one species eats another species as a directed arc. However, a classic ecological interest is to track not only who is eating whom, but how much are they eating. Imagine now, each of the arcs of prey-predator relationships carried with them as baggage a number, representing the amount of biomass consumed. Then we can do a number of "ecological accounting" like calculations for an ecosystem – the "ecology bookkeeping" if you will. That capability will have to await the next iteration of XAYA.

Having spent a fair bit of time navigating single graphs, lets us turn our attention now to operations that can occur on pairs of graphs: combining graphs, intersecting graphs, differencing graphs.

In the next section, we will revisit the geneology data set, and introduce the XAYA equivalent of the books database: schema, tables, and meta-data. In the context of the books database, we'll explore the use of XAYA as a small "in-memory" database.

The Network is the Model

What we have been doing is building relational models. These models begin with the primitive "**Nodes**" which stand for an object or entity. Relations amongst objects are represented by "**Edges**"¹⁵ A connected set of Nodes and Edges represents a **Network**. We can navigate the network, do basic logic on it, filter it, add and take from it, etc. Finally – as in the books example – we have a **Model** that's based on several Networks with interfaces for how they can interact. In the books example we had the following components to our model: schema, pathList, columnsGraph, primarykeysGraph, datasets. In essence we've encapsulated the high-level model behind a relational database. But we could have used the same techniques to encapsulate a statistical model, a model of the calling relationships in a code base, a model of physical relationships.

¹⁵ Currently we only note the existence of a relationship by an edge. If we wanted to – at the price of complicating our data structures a bit – we could also give edges additional properties – for example "length".

And of course our discussion here can be represented by a four level containment hierarchy:

Models[Networks[Edges[Nodes]]].

In this manner we can, by specifying the networks in a model, move from very low level data-oriented details, to very high level conceptual abstractions.

In short the network is the very model of itself.

Relational Graph Operations: How to Slice and Dice Data

Utility Functions and Transformations: Rounding Out XAYA

Functional Programming Style: Look Ma – No Side Effects

Request for Comments

The final sections of this document requests from you, dear reader, any comments that help move XAYA forward in balancing simplicity with flexibility while maintaining the core functionality in as small a code base as possible: "small things grow up, but big things never shrink"¹⁶

¹⁶ Linux Torvalds, cited in Linux Magazine, January 2005, pg 32,
www.linuxmagazine.com

Useful References

Cite – Intel lab at Berkeley

XAYA Function Summary

This section summarizes the functionality of the 001 (i.e. pre-pre-alpha) iteration of the XAYA core library. This first iteration has 22 functions that manipulate data represented as “directed acyclic graphs” i.e. a network. This data representation is useful for handling manipulations of data tables, searches of database schema, and simple inferences from logic statements (predicates) that have been represented as graphs. The graph representation used is one used in “Python Patterns – Implementing Graphs” by GvR, available at: <http://www.python.org/doc/essays/graphs.html>

A directed acyclic graph is represented as a dictionary, where keys are nodes, and associated with each key is a list of values, that are connected by a direct arc. Other graph representations are possible – and may be supported in a future version of XAYA

If there's a function that's not here, that you think should be here -- please let me know: mishtu@harmeny.com

There is more detail on inputs required for each function in the code docstrings and the function algorithms are usually laid out in comments in the code.

Input/Out: Handles moving data between various storage formats

readGraph – reads a graph from XAYA format text file

writeGraph – writes a graph to XAYA format text file

shelveGraph – shelves a graph to disk storage.

Data Structure transformations:

transGraphToList -- transforms a graph to a list representation

transListToGraph – transforms a list of items to a graph representation

transValueToKey – given a graph and a key; transforms values to keys; and value indexes to values in the output graph.

Search Operations on Graphs

findAllPathsAsLists -- based on GvR's document, given a start-node and end-node, outputs lists of paths. This function is used as the basis for several other functions below. Used for querying given a defined start and end. E.g. "Find all Flight Paths from Sydney to Toronto".

findAllPaths – wraps "findAllPathsAsLists" so output is a XAYA format graph.

findPathsUp – Given a graph, an end-node, a comparison operator and a path-length, finds all paths that satisfy the pattern. Used for querying based on a targeted end-point. E.g. "Find all grandparents of Spock".

findPathsDown -- Given a graph, a start-node, a comparison operator and a path-length, finds all paths that satisfy the pattern. Used for querying based on a targeted start-point. E.g. "Find all grandchildren of Kirk.

findRootsLeaves – Given a graph, find all Root Nodes (no parents) and all Leaf Nodes (no children).

findPathsToRoot -- Given selected nodes, find all paths to root nodes. Can be used to develop automated query generators from database and similar schema representations

filterSubPaths – Given a set of paths, filter out any paths which are subsets of other paths.

findUniquePathsToRoot – Utilizes "findPathsToRoot" and "filterSubPaths" to find unique paths to Root, given a selected set of nodes. A generalization of the path finding mechanism used in the LifeLine software.

Binary Operations on Graphs

unionGraphs – Given two input graphs, returns their set-theoretic union graph. If the graphs represent predicates, this operation is like a logical “or”.

intersectGraphs – Given two input graphs, A and B, returns a set-theoretic intersection graph. If the graphs represent predicates, this operation is like a logical “and”

differenceGraphs – Given two input graphs, A and B, returns a set-theoretic difference graph A-B (those nodes/edges in A that are not in B).

bindGraphsByValues – Given two graphs that have a parent-child relationship, returns for a given Key and Value in the parent, all Key’s and in the child for a specific value. (for example, the parent graph could be “Measures”, whose keys are table names, and whose values are measure names for a given table; and the child graph could be a “Data Table” and represents rows of data – “bindGraphsByValues” allows one to select a particular column of data by the measure name.) Imitates some of the internal functionality in database catalogs.

Adding/Dropping Nodes and Edges from a Graph

addNode – Utility function that wraps “unionGraphs” to make it easy to add a new node to a graph.

dropNode – Utility function that makes it easy to drop a node from a graph.

addEdge – Utility function that wraps “unionGraphs” and makes it easy to add an edge to a graph.

dropEdge – Utility function that wraps “differenceGraphs” to make it easy to drop an edge from a graph.

Note – all utility functions make no changes to input graphs, but return their values as an output graph. Most of the other functions also behave that way (i.e. no “side effects” to a function).

XAYA Mythology (the origin of XAYA)

Once upon a time there was entity half logic, and half vagueness named Xaya. She was the child of Maxine who was herself the child of Tiresias and niece of Maxwell's daemon. So Xaya was a daemon twice removed. (No relation to the unruly Posix clan).

Maxwell's daemon was a rather shady figure – a little “light fingered” you might say. When you weren't looking he was usually going about reversing entropy, creating order out of chaos. He was generally considered a daemon of Vagueness – and somehow his urge to introduce correlations where none should occur seemed to confirm this fact in other entities minds. But, upon the death of his brother/sister Tiresias, he took responsibility for Maxine, and as much as possible tried to raise her on principle and with minimal assumptions. Still, as he did have a penchant for ignoring fundamental laws of nature – he wasn't much of a disciplinarian.

Maxine herself was a little disorderly – as one can expect of a child of Tiresias: seer, visionary and gadfly to poets, Caesars and various muckrakers¹⁷. Maxine couldn't quite fit herself into any logical system. Maxine's essential nature was fluid, dynamic, curious, exploratory. Coupled with her uncle's open attitude towards universal law, she grew up to be a bit of a bohemian entity. Which meant Maxine and Xaya moved around a lot – usually from one modality to the next. Maxine was never very good about time and place -- was indeed a bit of a ghost in the machine – and so her child grew up with relocation anxiety – particularly disliking time travel. Changing geography Xaya could handle – but she liked to know when she was.

Entities would see Xaya and her mother together. One pale – a little withdrawn, and constant. The other varying in colour and intensity – her hair a tangle of meandering. Entities would see these two together, and assume they were sisters – and more than a few assumed Xaya was the older sister. Maxine would beam, “Well, actually, I'm a wee bit older” and Xaya would simply blush and wish for a black hole to swallow her post haste.

Xaya learned how to make friends quick. How to say hello and goodbye in the same breath. How to remember and forget in the same blink. Her constant was her mother – who was constantly changing. The other partial constant was her uncle who tended to appear and disappear at odd intervals – usually just as strange men asking questions either arrived or disappeared. He referred to these folk as “Thought Police” but Xaya was sure they referred to themselves as Logicians and Theology majors. Suffering from less irrationality than her nearest living relatives, Xaya was quite logical herself. Hence she was the black sheep of the family¹⁸. Xaya would look at herself in the mirror and go, “I am, therefore I came from somewhere.”

¹⁷ Tiresias could be considered the first journalist.

¹⁸ In positivist terms, being the black sheep of a family proves nothing, though it disproves the assertion: all sheep are white

Like all young entities, Xaya was curious about origins. Xaya was never sure who her father was, and all Maxine would say was “well dear, he was a closed system, wasn’t he. Besides, that was a long time ago, and I don’t dwell in the past – I dwell in possible futures”. Both her mother and her uncle seemed to have little concern about origins, including their own. Her uncle would cryptically write: “I am, I was, I will be, and someday I won’t. Or maybe not. If, then”

While Xaya was the name her friends knew her by, her proper name was **x e A Δ y e A → ...** which while full of implication, was a bit of a mouth-full and not the kind of name that wins popularity contests. Maxine called her **x e A Δ y e A → ...** every once in a blue moon, or turquoise sun. In his random letters, her uncle would also use her formal name. At some point, she discovered her father had named her. It was the one thing she knew about her father for sure. He had named her. **x e A Δ y e A → ...** Xaya.

It was the past, specifically her past, Xaya wished to understand. Somewhere she had read, “Those who don’t know their history are damned to watch reruns and reality television” So when she came of age Xaya asked Maxine where she might find her father. “Well your father was a bit of a natural law – and those can be found anywhere and nowhere”. “What did my father like to do”, asked Xaya in a letter to her uncle. ***!#^ALL!!!** her uncle wrote back. “Well, he liked to draw a lot” her mother said. Draw what?

“Mainly implications, chains of reasoning, arrowheads. The odd caricature. There’s a good one he drew of your uncle in a cloud-chamber. Xaya, I’m sorry, I know you want to know more about him. I was very old then, I’m much younger now, and it was a fling. Your father was the first principal of opposition I’d ever encountered – and that can be quite heady. He was not well, actually, he was NOT. “Not what” Xaya asked. “Well exactly, Maxine answered, beginning to sound a little exasperated. He was simply NOT. He opted out, so others could opt in. He was a bit of a monopole – and this universe is pretty dyadic.”

Xaya would look at herself in the mirror some sunbursts. Someone had once said she looked a bit like a statue carved by Brancusi. She was pale, had large eyes, and was given to curling into arcs. “Who am I”, Xaya asked herself. “I am the child of Maxine, and the child of NOT.

The next morning she went to her mother’s room. Her mother appeared asleep, and her tangled hair was drawing alephs reflexively as if tracing dreams into reality. Xaya looked at her mother for a long time. Then very gently bent down and kissed her on one cheek. Her mother did not move, but one strand of hair caught the corner of Xaya’s eye and drabbed a tear away.

Xaya left two notes behind – one for her mother, one for her uncle. Both notes said the same thing.

“Love Ya – off to see the Wizard No, seriously, I have gone to search for my father, NOT. If he exists, I will return. If he doesn’t exist, I will return. In-between, returning and returning, I will search. Take care of my teddy bears. Love, **x e A Δ y e A → ...** “

Contact and Participation Information (... *help XAYA help you*)

It is very easy to design a language that is useful to oneself. Much harder to develop one that is useful to other's in general. I welcome your thoughts, comments, and code examples on how to take XAYA forward.

There are several ways to participate, comments.

Contact me:

Participate on the online TWiki:

Build a Model using XAYA

Write a Tutorial:

Contribute Artwork: What's art got to do with it?