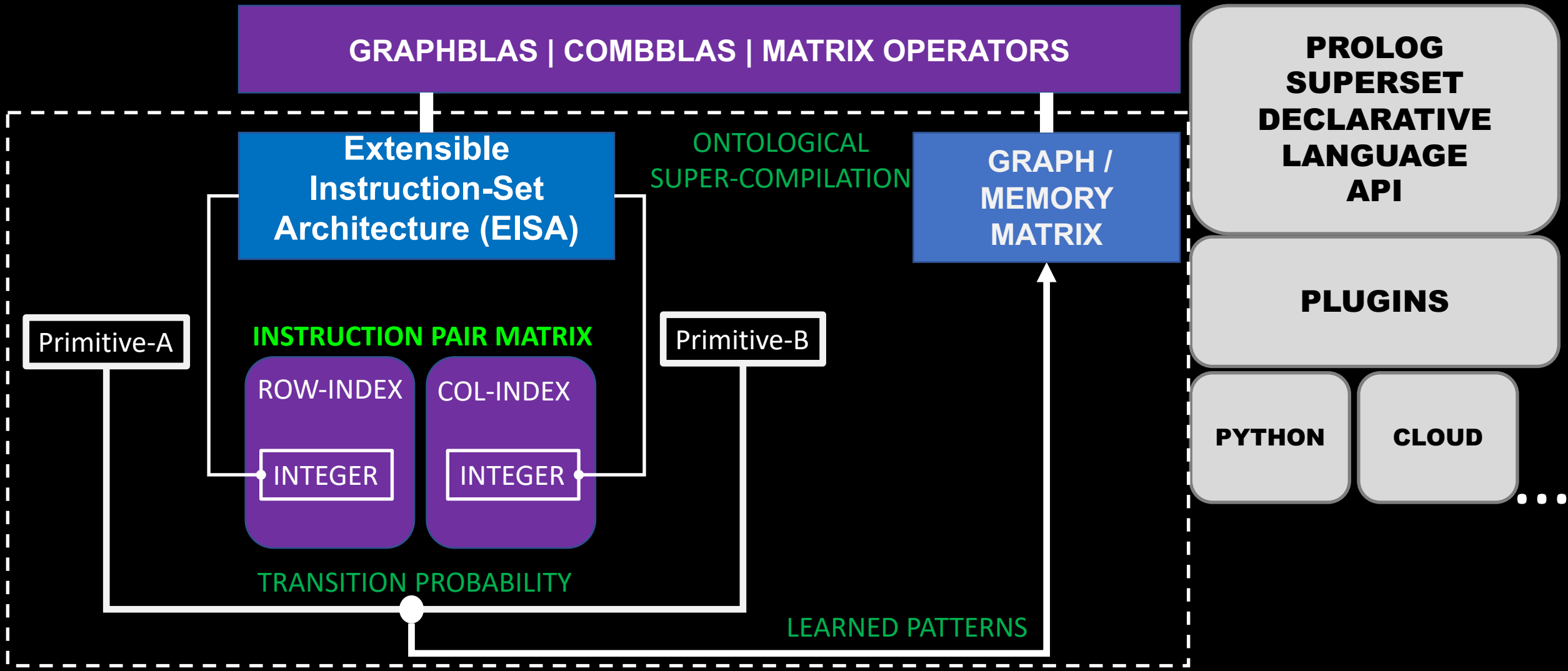
Ontology From Graph Signal Processing

John Sowa & Arun Majumdar

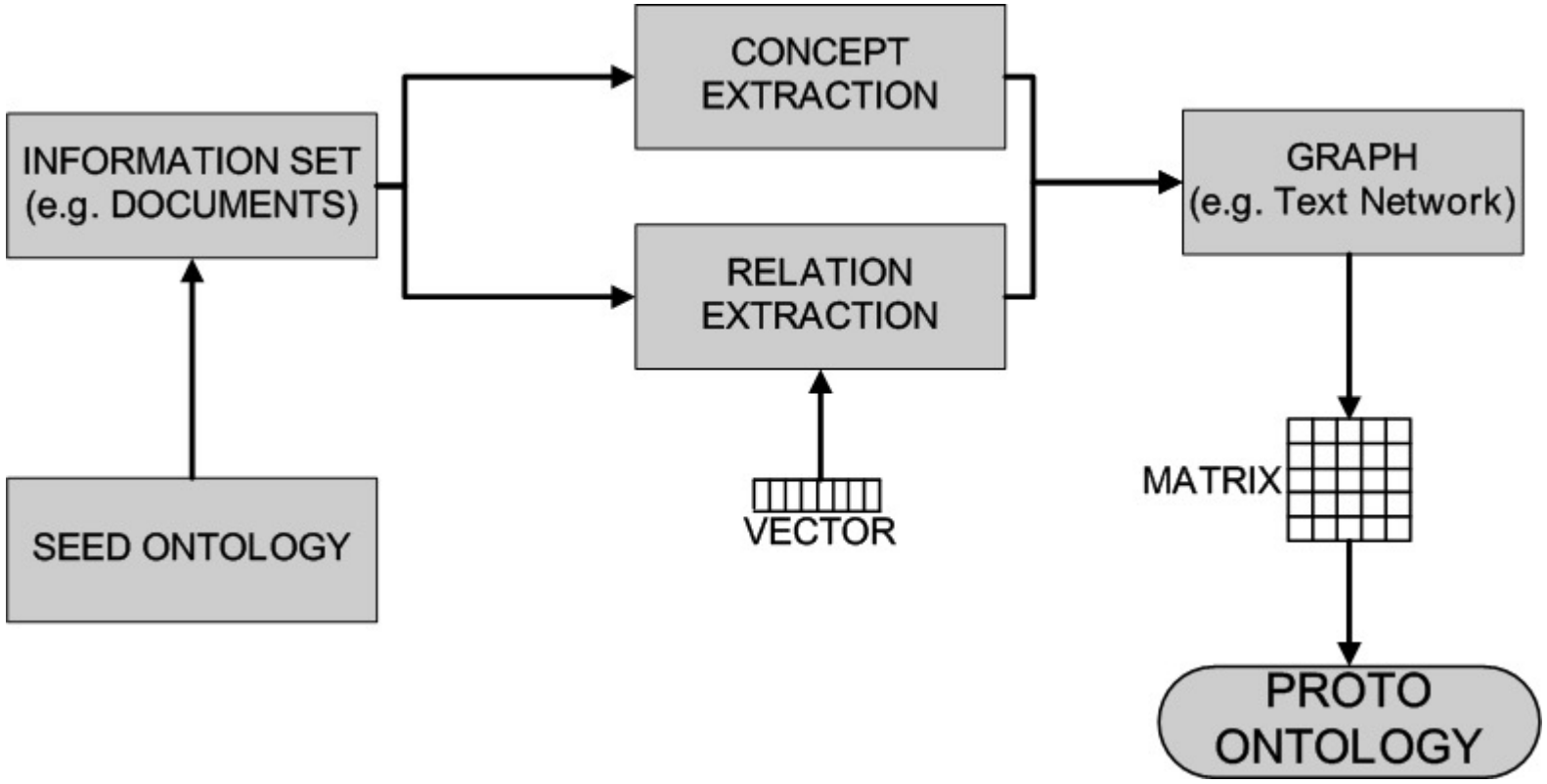
GraphStax LLC

& Neurosymbolics Inc.

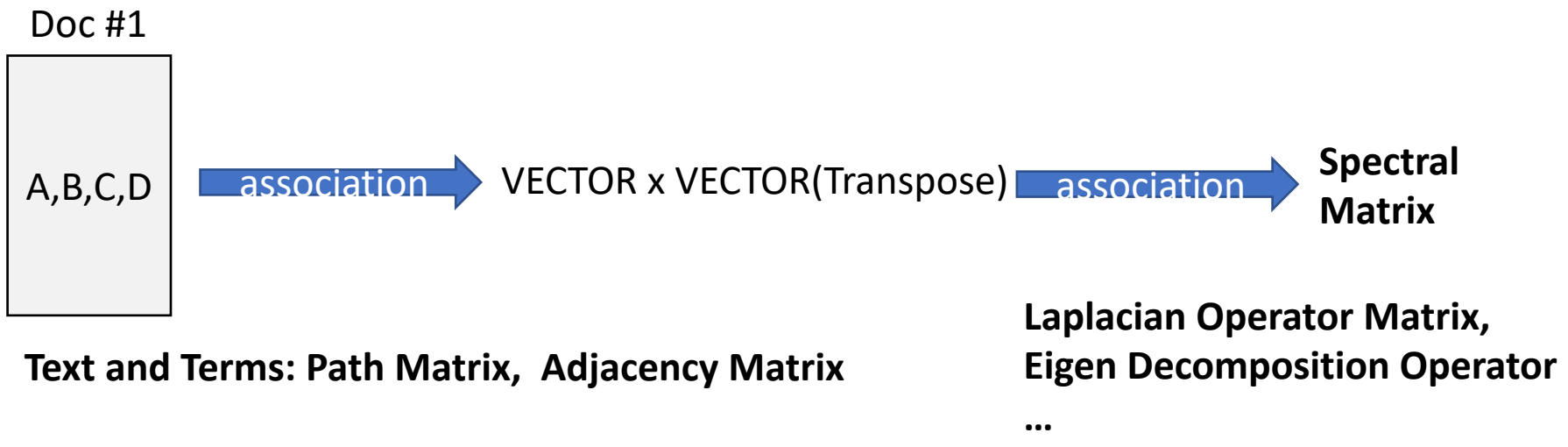
Neuro-Symbolic Logic Virtual Machine



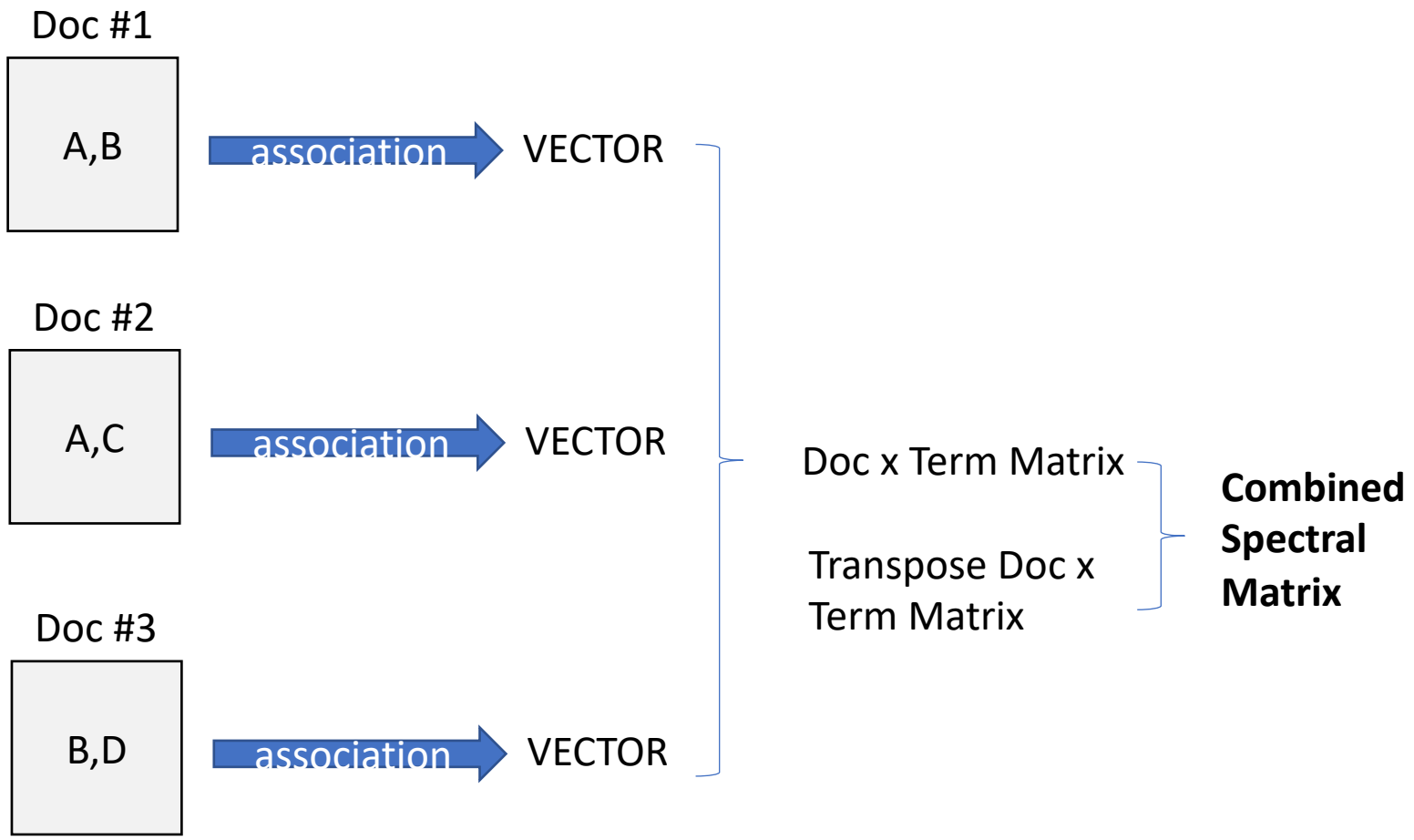
Architecture Overview



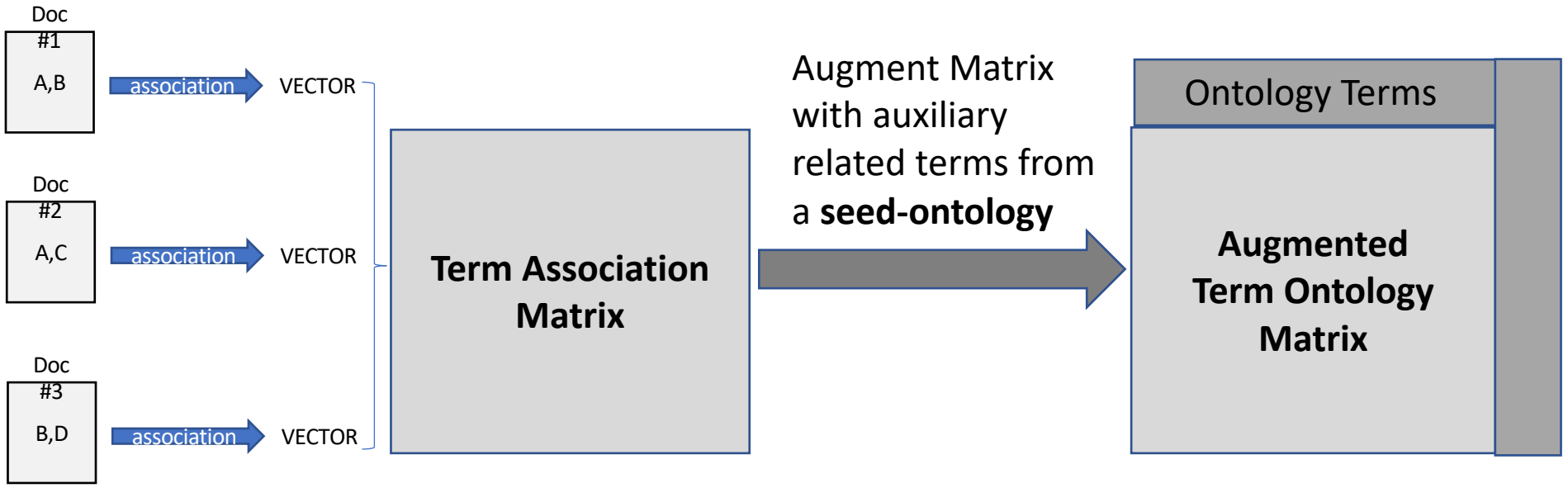
One Document



Many Documents = Combined Matrix



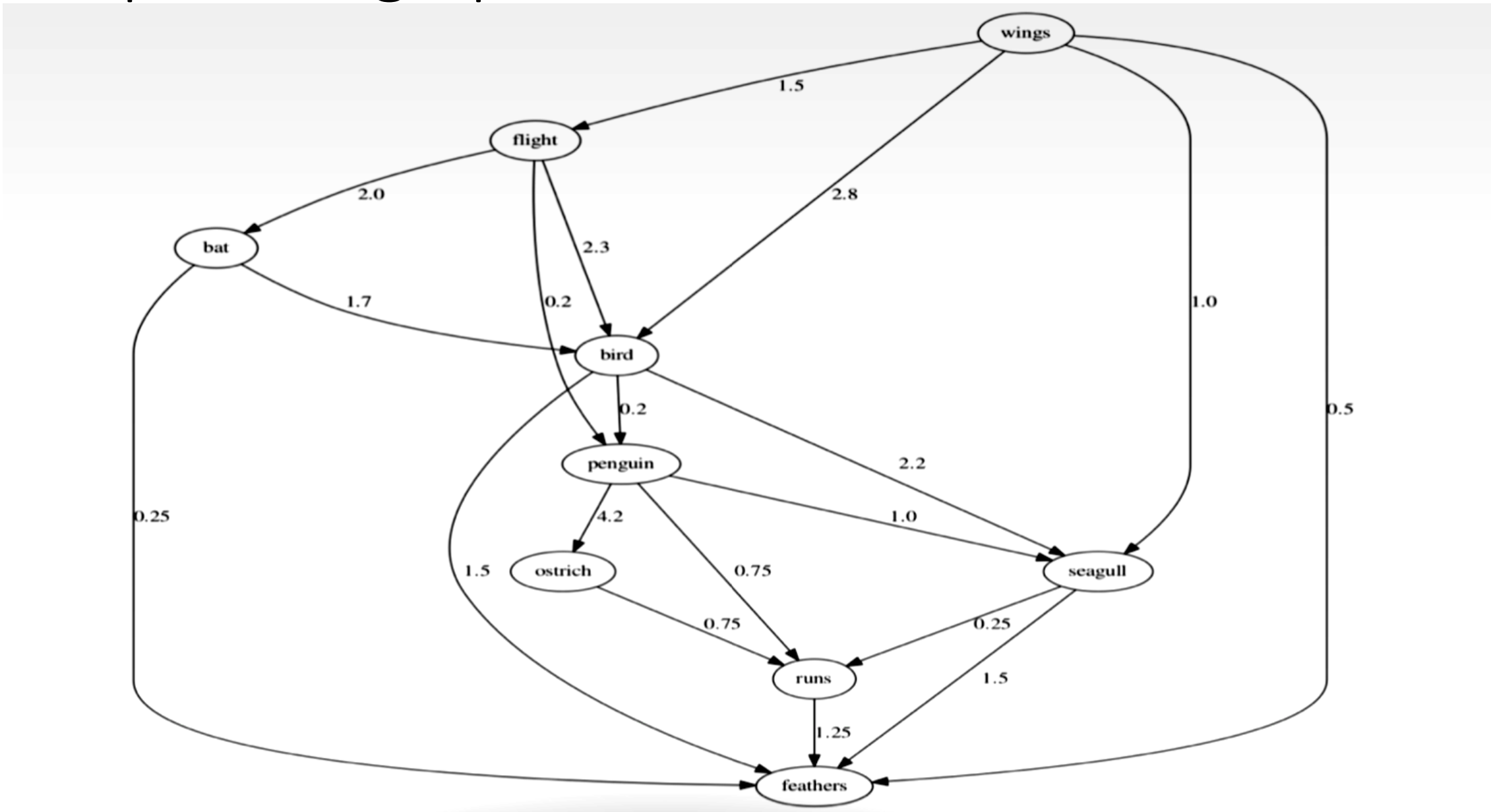
Human In The Loop



Resources for Semi-Automatic Extensions

Ontology	Entities	Size
KnowItAll	n/a	29,835
SUMO	20,000	70,000
WordNet	117,659	821,492
Cyc	300,000	3,000,000
TextRunner	n/a	7,800,000
DBpedia	3,500,000	672,000,000

A simple text graph



The Lexicographically Ordered Text Path Matrix

```
              bat  bird  feathers flight . . .
bat          [[1.   1.7  0.25 2.   0.   0.   0.   0.   0.  ]
bird         [1.7  1.   1.5  2.3  0.   0.2  0.   2.2  2.8 ]
feathers     [0.25 1.5  1.   0.   0.   0.   1.25 1.5  0.5 ]
flight       [2.   2.3  0.   1.   0.   0.2  0.   0.   1.5 ]
     .       [0.   0.   0.   0.   1.   4.2  0.75 0.   0.  ]
     .       [0.   0.2  0.   0.2  4.2  1.   0.75 1.   0.  ]
     .       [0.   0.   1.25 0.   0.75 0.75 1.   0.25 0.  ]
             [0.   2.2  1.5  0.   0.   1.   0.25 1.   1.  ]
             [0.   2.8  0.5  1.5  0.   0.   0.   1.   1.  ]]
```

Set Up Matrix

```python
import os, sys
import math
import operator
import numpy
import pylab
import networkx
import matplotlib.pyplot as pylab
import scipy.sparse
from scipy import linalg, matrix
from sympy import Matrix, sympify, S, symbols
from sympy.concrete.summations import summation
from sympy.physics import secondquant as sq

import g

# measure from a concept to "itself" in the path measure matrix
UNIT_ENTRY = 1
# MATRIX_EXPONENTIALS_LIMIT: limit to the number of powers taken
MATRIX_EXPONENTIALS_LIMIT = 10

#   Distance approach:
#    1.) init with concepts and relations (see unittest):  semantic_network = DistanceMatrixRepresentation(CONCEPTS, links); links: the list of relations and their directional measures between concepts
#    2.) Generate the Ranking-Operator via eigen-decomposition:  semantic_network.ranking_operator_matrix()
#    3.) Compute the field energies between semantics:  semantic_network.apply_ranking_operator(["a", "b"])
class DistanceMatrixRepresentation(object):
    def __init__(self, links, logger=None):
        self.concepts = []
        self.logger = logger

        # initialize concepts from relations
        for link in links:
            a, b, v = link            # use the values found in links
            self.concepts.append(a)
            self.concepts.append(b)
        self.concepts = sorted(set(self.concepts))
        self.cardinality = len(self.concepts) # num of concepts is size of matrix
        # create a dict of "concept" -> index -- for easy access to stuff
        #self.cindex = dict( (i,c) for c,i in enumerate(concepts))
        self.cindex = dict( (concept,index) for index,concept in enumerate(self.concepts))
        # create a symmetric path matrix and normalize
        self.normed_matrix(links)
```

Set up a Matrix of Terms Sorted Lexicographically (for indexing)

Set up a Matrix of Distances between Terms (for ranking)

Normalize using the Determinant

```
43      # Create the Ranking-Operator using eigen decomposition perform, if selected, a reduction of dimensions by largest values
44      # normalizes the evience measures to form the unitary bistochastic matrix
45      def ranking_operator_matrix(self, reduce_to_dims=None, final_normalize=False):
46          self.FINAL_NORMALIZE = final_normalize
47          self.HEURISTIC_DIMENSIONALITY_REDUCTION=False
48          if reduce_to_dims:
49                  self.HEURISTIC_DIMENSIONALITY_REDUCTION = True
50                  self.REDUCED_DIMS = reduce_to_dims
51          self.solve_for_ranking_operator()
52
53      # create a symmetric (real or complex) measure path-matrix from the input text-graph (conceptual graph);
54      # Normalize the matrix using the determinant, resulting in a bistochastic matrix (probability measures).
55      # When two concepts are connected by an assertion in the "semantic Prolog database" (e.g. bigrams), insert their
56      # directed probability measures in the two appropriate entries of the matrix. All remaining entries are 0.
57      # See:  https://en.wikipedia.org/wiki/Doubly_stochastic_matrix
58      # See:  https://en.wikipedia.org/wiki/Renormalization
59
60      def normed_matrix(self, links):
61          self.PathMatrix = numpy.matrix( numpy.zeros( shape=(self.cardinality,self.cardinality) ) )
62          # initialize diagonal values
63          for i in range(self.cardinality):
64              self.PathMatrix[i,i] = UNIT_ENTRY
65          print("\n\nInitialized matrix:\n", self.PathMatrix)
66          # initialize matrix with concept relations
67          for l in links:
68              s, o, p = l
69              # use the values found in links
70              self.PathMatrix[self.cindex[s], self.cindex[o]] = p
71              self.PathMatrix[self.cindex[o], self.cindex[s]] = p
72
73          print("\nPath matrix from graph data:\n", self.PathMatrix)
74          # make unit vectors using determinant
75          self._normalizer()
76          print("\nNormed Graph Data Path Matrix:\n", self.PathMatrix)
```

Solve using Eigendecomposition

```
    def solve_for_ranking_operator(self):
        # eigenvectors and values for self.PathMatrix
        eigenvalues,eigenvectors = numpy.linalg.eigh(self.PathMatrix)
        print("\n\nEigenvalues for self.PathMatrix", eigenvalues)
        print("\n\nEigenvectors for self.PathMatrix", eigenvectors)
        if self.FINAL_NORMALIZE:
            eigenvalues = eigenvalues / 2 # tmp, test
        if self.HEURISTIC_DIMENSIONALITY_REDUCTION:
            eigenvalues, eigenvectors = self._dim_reduce(eigenvalues, eigenvectors)
            dimension_sizes = self.REDUCED_DIMS*2
        else:
            dimension_sizes = self.cardinality
        # create diagonal matrix of eigenvalues
        diagonal_matrix_of_eigenvalues = numpy.matrix( numpy.zeros( shape=(dimension_sizes, dimension_sizes) ) )
        for i in range(dimension_sizes):
            diagonal_matrix_of_eigenvalues[i,i] = eigenvalues[i]
        #print("\n\ndiagonal matrix of eigenvalues\n",diagonal_matrix_of_eigenvalues)
        MatrixExponential = diagonal_matrix_of_eigenvalues
        current_El = diagonal_matrix_of_eigenvalues.copy()
        for i in range(2,MATRIX_EXPONENTIALS_LIMIT+1):
            current_El = numpy.dot(diagonal_matrix_of_eigenvalues, current_El)
            # for binary systems, pow(2,i), for, arbitrary perms factorial(i)
            MatrixExponential += ( current_El / math.factorial(i) )
        print("\nMatrixExponential\n", MatrixExponential)
        # Ranking operator is  (V*L*Vt); so compute: V * MatrixExponential * V.T
        if self.FINAL_NORMALIZE:
            Vs = eigenvectors * MatrixExponential
            Vss = self.normed_vectors(Vs)
            self.ranking_operator = Vss * Vss.T
        else:
            #default
            self.ranking_operator = eigenvectors * MatrixExponential * eigenvectors.T + 1
        print("\n\nRanking_operator\n", self.ranking_operator)
        return "Ranking-Operator Completed"

# reduce the dimensions of eigenvalues and eigenvectors
# Retain (concatenate) REDUCED_DIMS of lowest and highest eigenvalues and eigenvectors
    def _dim_reduce(self, eigenvalues, eigenvectors):
        reduc_evals =  numpy.concatenate( (eigenvalues[:self.REDUCED_DIMS], eigenvalues[-self.REDUCED_DIMS:]), axis=1 )
        reduc_evecs =  numpy.concatenate( (eigenvectors[:,:self.REDUCED_DIMS], eigenvectors[:,-self.REDUCED_DIMS:]), axis=1)
```

Here is the Ranking Operator

```
Ranking_operator
 [[0.02163871 0.02702995 0.01004569 0.02347979 0.00049209 0.00271022
  0.00143471 0.01328967 0.02286122]
 [0.02702995 0.06821484 0.02585235 0.03861012 0.0038109  0.00983807
  0.00778729 0.03154862 0.03905455]
 [0.01004569 0.02585235 0.02026239 0.01504925 0.00378538 0.00817085
  0.00806952 0.02073618 0.02011635]
 [0.02347979 0.03861012 0.01504925 0.03544032 0.00251705 0.0038835
  0.00151275 0.02082392 0.02889711]
 [0.00049209 0.0038109  0.00378538 0.00251705 0.04819012 0.02991142
  0.01316256 0.01167059 0.00114248]
 [0.00271022 0.00983807 0.00817085 0.0038835  0.02991142 0.05111524
  0.01403011 0.0099146  0.00604607]
 [0.00143471 0.00778729 0.00806952 0.00151275 0.01316256 0.01403011
  0.00973211 0.00902036 0.00348782]
 [0.01328967 0.03154862 0.02073618 0.02082392 0.01167059 0.0099146
  0.00902036 0.02831261 0.02636723]
 [0.02286122 0.03905455 0.02011635 0.02889711 0.00114248 0.00604607
  0.00348782 0.02636723 0.03524017]]
```

Output Ranks (Fiedler Vector Ordering)

Ranking from bird to **flight (greatest to least)**
[('**bird**', 0.1068249657423646 1), ('flight', 0.0740504430826141 6), ('wings', 0.0679516601973721 4), ('seagull', 0.0523725391223734 1), ('bat', 0.0505097329128958 44), ('feathers', 0.0409015994256867 26), ('penguin', 0.0137215694709869 38), ('runs', 0.0093000381042825 84), ('**ostrich**', 0.0063279432684448 71)]

Ranking from **bird (greatest to least)**
[('**bird**', 0.0682148432963939 8), ('wings', 0.0390545482703763 6), ('flight', 0.0386101224459706 3), ('seagull', 0.0315486239040213 6), ('bat', 0.0270299465346640 8), ('feathers', 0.0258523542469130 36), ('penguin', 0.0098380694724898 39), ('runs', 0.0077872915061591 84), ('**ostrich**', 0.0038108978371336 23)]

Ranking from **penguin (greatest to least)**
[('**penguin**', 0.0511152408030028 8), ('ostrich', 0.0299114228068219 22), ('runs', 0.0140301059820932 57), ('seagull', 0.0099145990349793 97), ('bird', 0.0098380694724898 39), ('feathers', 0.0081708469111617 14), ('wings', 0.0060460692109726 45), ('flight', 0.0038834999849 71), ('**bat**', 0.0027102214155942748)]
.

Thank you!

- Questions?