Networks Analysis in Julia

Follow along code https://github.com/nassarhuda/CSE17

Huda Nassar
Computer Science
Purdue

with David F. Gleich
Computer Science
Purdue

I'm supported by
DARPA SIMPLEX
& more broadly NSF,
DOE, NASA, Sloan

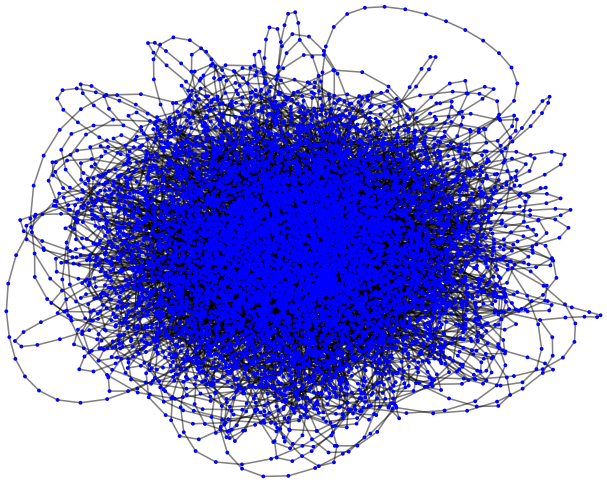
Many operations we want to perform on networks are (in fact) linear algebra

Network Operation	Matrix Operation
Ranking (eg. PageRank)	$(\boldsymbol{I} - \alpha \boldsymbol{P})\mathbf{x} = (1-\alpha)\mathbf{v}$
Network Alignment (eg. IsoRank)	$\mathbf{x} = \alpha(\boldsymbol{A} \otimes \boldsymbol{B})\mathbf{x} + (1-\alpha)\mathbf{h}$
Diffusion (eg. HeatKernel)	$\mathbf{x} = \exp(-t(\boldsymbol{I} - \boldsymbol{P}))\,\mathbf{s}$
Community Detection (eg. Random walks)	$\mathbf{x} = \boldsymbol{P}^k\mathbf{s}$
Clustering (eg. Spectral Clustering)	$\boldsymbol{L} = \boldsymbol{I} - \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$

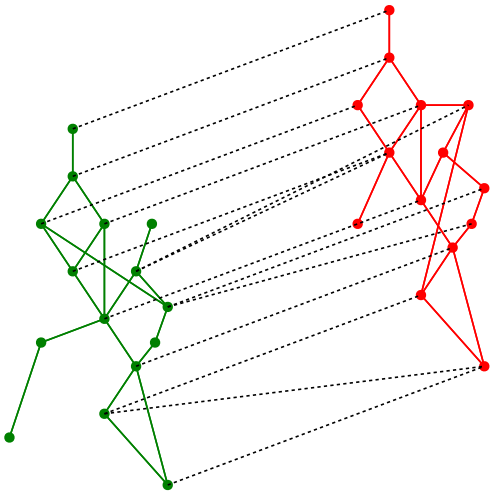
These give simple Julia implementations

```
x = (eye(n) - alpha*P) \ (1-alpha)*v  # PageRank
x = expm(-t*(eye(n)-P))*v             # Heat Kernel
(That we can also improve upon ...)
```

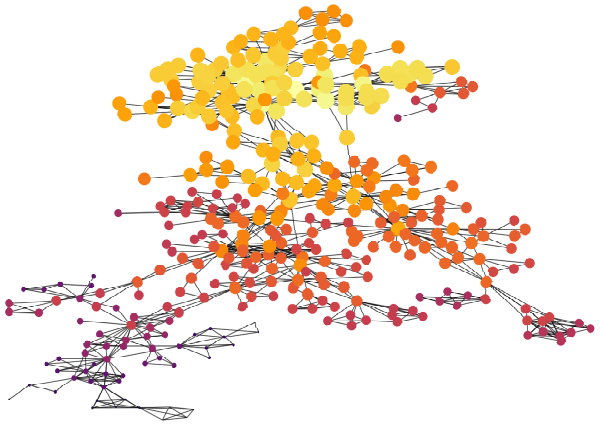
We used Julia in these three research projects



generation of large graphs
(1 Billion nodes)



network alignment

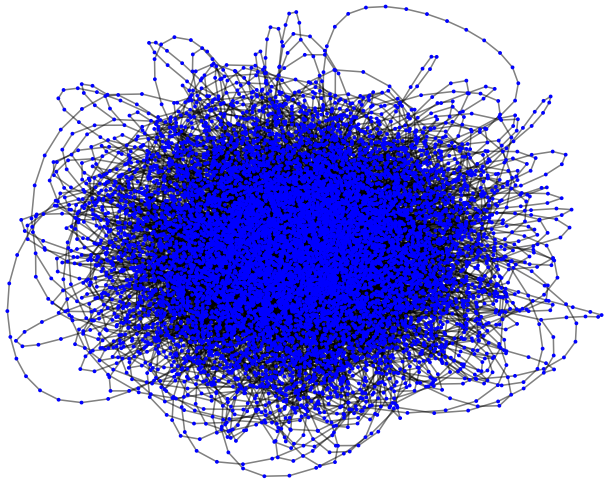


link prediction

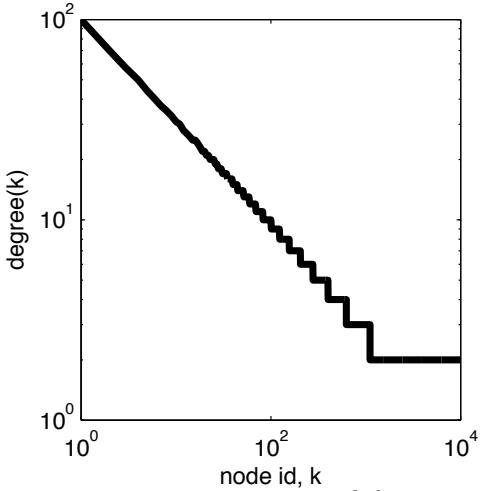
Julia's wrappers make it easy to use existing C code

Goal: want to generate large graphs satisfying a given degree distribution to study their localization behavior

need to generate a graph:



under a powerlaw degree distribution



such that

$$d(k) \leq \max(dk^{-p}, \delta)$$

Strong Localization in Personalized PageRank
Nassar, Kloster, Gleich, WAW2015
Localization in seeded PageRank (submitted

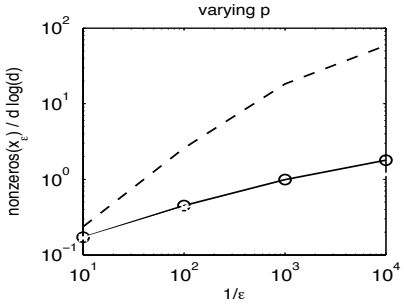
Julia's wrappers make it easy to use existing C code

Previously,

run
`c code`

save it to the hard disk

read it into matlab



run experiments

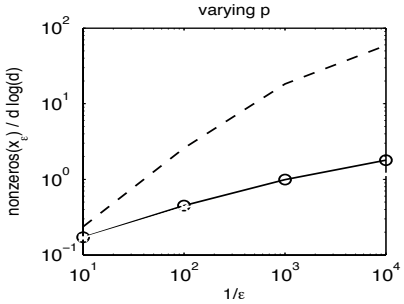
Julia's wrappers make it easy to use existing C code

Previously,

run
`c code`

save it to the hard disk

read it into matlab



run experiments

file size ~ 40GB
when network is large

O(n) algorithm for compute time

too much time!

Julia's wrappers make it easy to use existing C code

The Julia solution:

1. use existing C code
2. use Julia function call of the form

```
ccall ( (:check_graphical_sequence, libpath),
          Cint, # return value
          (Int64, Ptr{Int64}), # arg types
          n, degs); # actual args
```

3. wrap inside a Julia function

Result: Can generate a graph with 1B nodes and 2B+ edges in less than an hour

nodes/edges	time
1K/2K+	0.001 secs
1M/2M+	5 secs
100M/200M+	260 secs
1B/2B+	3570 secs

Julia's wrappers make it easy to use existing C code

To compare the the 2 approaches

With Julia

nodes/edges	time
1K/2K+	0.001 secs
1M/2M+	5 secs
100M/200M+	260 secs
1B/2B+	3570 secs

Previously,

nodes/edges	time
1K/2K+	0.2 secs (200x)
1M/2M+	41 secs (10x)
100M/200M+	3300 secs (8x)
1B/2B+	9941 secs (2.7 hours) (3x)

Julia's wrappers make it easy to use existing C code

To regenerate these results:
julia:

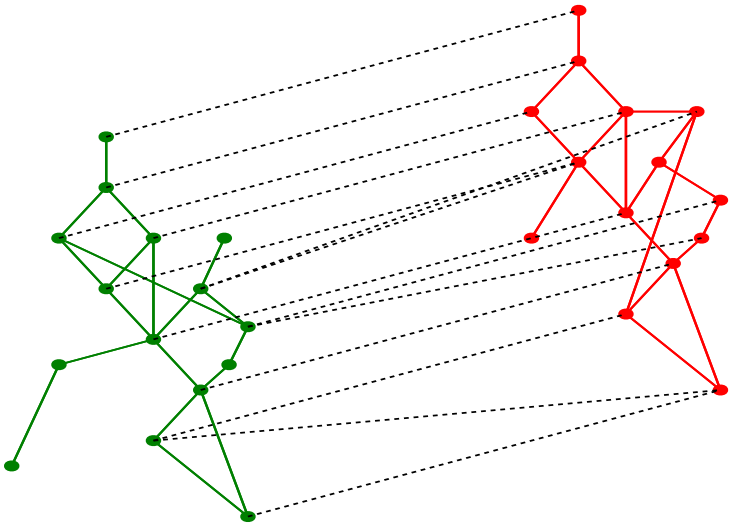
```
$ cd bisquik-julia-wrapper
$ make clean; make
julia> include("create_graph.jl")
julia> tic();create_graph(p,n,dmax,dmin); toc()
```

matlab:

```
$ cd bisquik-master
$ make clean; make
$ cd../power_law_analysis
>> tic;create_graph(p,n,dmax,dmin);toc;
```

Julia has built in features that make it easy to manipulate large sparse matrices

Goal: want to implement various state-of-the-art network alignment algorithms. One problem boils down to solving a bipartite matching problem on each row of the matrix



What is the best way of matching graph A (green) to graph B (red) using edges in L (dashed lines)?

Julia has built in features that make it easy to manipulate large sparse matrices

Previously,

```
#include "mex.h"


#if MX_API_VER < 0x07030000
typedef int mwIndex;
typedef int mwSize;
#endif // MX_API_VER

…
```

compile with matlab's `mex`

Julia has built in features that make it easy to manipulate large sparse matrices

Previously,

```
#include "mex.h"


#if MX_API_VER < 0x07030000
typedef int mwIndex;
typedef int mwSize;
#endif // MX_API_VER

…
```

compile with matlab's `mex`

buggy
interface is harder
weak compatibility and reproducibility
inconsistent indexing

Julia has built in features that make it easy to manipulate large sparse matrices

The Julia solution:

1. Simply re-write the entire code in native Julia
    1. Make use of any Julia feature available
    2. Write type-stable julia code (will return to this in 2 slides)

Result: Our Julia implementation beats the C++ implementation in most cases (both very well written)

Problem size	time
12 nodes	0.003 secs
16952 nodes	0.18 secs
4971629 nodes	40 secs

Julia has built in features that make it easy to manipulate large sparse matrices

To compare the the 2 approaches

With Julia

Problem size	time
12 nodes	0.003 secs
16952 nodes	0.18 secs
4971629 nodes	40 secs

Previously,

Problem size	time
12 nodes	0.018 secs
16952 nodes	0.24 secs
4971629 nodes	50 secs

notes on re-generating these results can be found in:
/netalignmr/

Here are some performance tips…

```
type all_matching_output
    q::Vector{Float64}
    mi::Vector{Int64}
    mj::Vector{Int64}
    medges::Int64
end
```

Use types liberally.

```
Qp = Qt.colptr
Qr = Qt.rowval
Qv = Qt.nzval
```

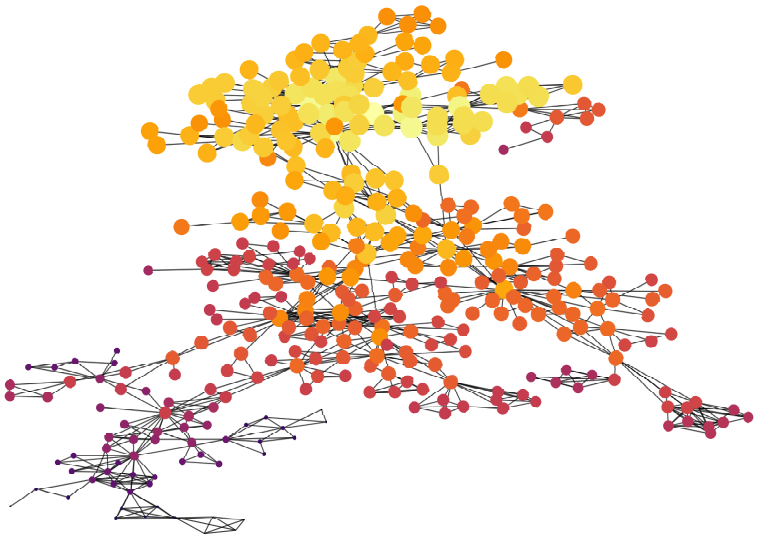
Make use of Julia's internal representations.

```
l1 = zeros(Float64,n)
l2 = zeros(Float64,n+m)
s = ones(Int64,n+m)
t = -1*ones(Int64,n+m)
offset = zeros(Int64,n)
```

Pre-allocate and declare types.

Julia is easily parallelizable

Goal: need to use an existing link prediction method (aptrank) that is iterative and requires a matrix-matrix product in each iteration.



An Adaptive PageRank Model for Protein Function Prediction on Bi-relational Graph

B. Jiang, K. Kloster, D.F. Gleich, M. Gribskov

The idea of AptRank is derived from seeded PageRank where we wish to learn information about a certain node in the network (in this case predict links)

$$\left(\begin{bmatrix} \boldsymbol{I}_m & 0 \\ 0 & \boldsymbol{I}_n \end{bmatrix} - \alpha \begin{bmatrix} \boldsymbol{G} & \boldsymbol{R} \\ \boldsymbol{R}^T & \boldsymbol{H} \end{bmatrix} \right) \begin{bmatrix} \boldsymbol{X}_G \\ \boldsymbol{X}_H \end{bmatrix} = (1-\alpha) \begin{bmatrix} \boldsymbol{I}_m \\ 0 \end{bmatrix}$$

AptRank learns the alpha to use at each iteration

Julia is easily parallelizable

Previously,

Used Matlab native implementation with a parallel matrix matrix product operation

```
Zc = cell(1,nblocks);
for i = 1:nblocks
    start = 1 + (i-1)*bs;
    subset = start:min(start+bs-1,n);
    Zc{i} = Z(:,subset);
end % slice Z

Tc = cell(1,nblocks);
parfor bi = 1:nblocks
    Tc{bi} = P*Zc{bi};
end
T = cell2mat(Tc);
```

Julia is easily parallelizable

The Julia code snippet:

```
for i = 1:np
    t = Z[:,all_ranges[i]]
    sendto(i,C=t)
end

…

@everywhere C = P*C

…

for i = 1:np
    Ci = getfrom(i, :C)
    Zh[all_ranges[i],:] = Ci[m+1:size(Zt,2),:]'
end
…
```

Julia is easily parallelizable

To compare the the 2 approaches (12 procs)

With Julia

Previously,

Problem	Julia	MATLAB
Problem size: 3904x1695	270 secs	295 secs
Problem size: 13281x2919	1365 secs	1394 secs
Problem size: 13562x2919	7174 secs	7923 secs

notes on re-generating these results can be found in:
/aptrank/regenerate_aptrank.txt

Packages in Julia

Many Julia Packages exist to help researchers with their work:

Ours:
MatrixNetworks.jl (https://github.com/nassarhuda/MatrixNetworks.jl)
NetworkAlign.jl (https://github.com/nassarhuda/NetworkAlign.jl)

Others:
Convex Optimization (Convex.jl)
Plots (Plots and PyPlots)
Interface to call Python functions (PyCall.jl)

More:
http://pkg.julialang.org/

Thank you!

Huda Nassar
email: hnassar@purdue.edu
@nassarhuda