End-to-End In-memory Graph Analytics

Jure Leskovec (@jure)

Including joint work with Rok Sosic, Deepak Narayanan, Yonathan Perez, et al.
Background & Motivation

My research at Stanford:

- Mining large social and information networks
- We work with data from Facebook, Twitter, LinkedIn, Wikipedia, StackOverflow

Much research on graph processing systems but we don’t find it that useful…

Why is that? What tools do we use? What do we see are some big challenges?
Some Observations

- We do not develop experimental systems to compete on benchmarks
  - BFS, PageRank, Triangle counting, etc.
- Our work is
  - Knowledge discovery: Working on new problems using novel datasets to extract new knowledge
  - And as a side effect developing (graph) algorithms and software systems
End-to-End Graph Analytics

Need end-to-end graph analytics system that is flexible, scalable, and allows for easy implementation of new algorithms.
Typical Workload

1. Finding experts on Stack Overflow:
   - Posts
     - Select
   - Questions
     - Join
   - Answers
     - Select
   - Python Q&A
     - Join
     - Construct Graph
     - SNAP
     - PageRank Algorithm
     - Scores
       - Join
     - Experts
     - Users

Jure Leskovec, Stanford
Observation

Graphs are never given!
Graphs have to be constructed from input data! (graph constructions is a part of knowledge discovery process)

Examples:

- Facebook graphs: Friend, Communication, Poke, Co-tag, Co-location, Co-event
- Cellphone/Email graphs: How many calls?
- Biology: P2P, Gene interaction networks
Graph Analytics Workflow

Input: Structured data
Output: Results of network analyses
  - Node, edge, network properties
  - Expanded relational tables
  - Networks

Raw data
video, text, sound, events, sensor data, gene sequences, documents, ...

Structured data
Relational tables

Graph analytics

Hadoop
MapReduce
Plan for the Talk: Three Topics

- **SNAP**: an in-memory system for end-to-end graph analytics
  - Constructing graphs from data

- Multimodal networks
  - Representing richer types of graphs

- New graph algorithms
  - Higher-order network partitioning
  - Feature learning in networks
SNAP
Stanford Network Analysis Platform


End-to-End Graph Analytics

- **Stanford Network Analysis Platform (SNAP)**
  General-purpose, high-performance system for analysis and manipulation of networks
  - C++, Python (BSD, open source)
  - [http://snap.stanford.edu](http://snap.stanford.edu)
- Scales to networks with hundreds of millions of nodes and billions of edges

New knowledge and insights
Desiderata for Graph Analytics

- Easy to use front-end
  - Common high-level programming language
- Fast execution times
  - Interactive use (as opposed to batch use)
- Ability to process large graphs
  - Billions of edges
- Support for several data representations
  - Transformations between tables and graphs
- Large number of graph algorithms
  - Straightforward to use
- Workflow management and reproducibility
  - Provenance

Jure Leskovec, Stanford
Data Sizes in Network Analytics

<table>
<thead>
<tr>
<th>Number of Edges</th>
<th>Number of Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;0.1M</td>
<td>16</td>
</tr>
<tr>
<td>0.1M – 1M</td>
<td>25</td>
</tr>
<tr>
<td>1M – 10M</td>
<td>17</td>
</tr>
<tr>
<td>10M – 100M</td>
<td>7</td>
</tr>
<tr>
<td>100M – 1B</td>
<td>5</td>
</tr>
<tr>
<td>&gt; 1B</td>
<td>1</td>
</tr>
</tbody>
</table>

- Networks in Stanford Large Network Collection
  - [http://snap.stanford.edu](http://snap.stanford.edu)
  - Common benchmark Twitter2010 graph has 1.5B edges, requires 13.2GB RAM in SNAP
## Network of all Published research

<table>
<thead>
<tr>
<th>Entity</th>
<th>#Items</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Papers</td>
<td>122.7M</td>
<td>32.4GB</td>
</tr>
<tr>
<td>Authors</td>
<td>123.1M</td>
<td>3.1GB</td>
</tr>
<tr>
<td>References</td>
<td>757.5M</td>
<td>14.4GB</td>
</tr>
<tr>
<td>Affiliations</td>
<td>325.4M</td>
<td>15.3GB</td>
</tr>
<tr>
<td>Keywords</td>
<td>176.8M</td>
<td>5.9GB</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1.9B</strong></td>
<td><strong>104.1GB</strong></td>
</tr>
</tbody>
</table>

- Microsoft Academic Graph
<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Items</th>
<th>Raw Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>DisGeNet</td>
<td>30K</td>
<td>10MB</td>
</tr>
<tr>
<td>STRING</td>
<td>10M</td>
<td>1TB</td>
</tr>
<tr>
<td>OMIM</td>
<td>25K</td>
<td>100MB</td>
</tr>
<tr>
<td>CTD</td>
<td>55K</td>
<td>1.2GB</td>
</tr>
<tr>
<td>HPRD</td>
<td>30K</td>
<td>30MB</td>
</tr>
<tr>
<td>BioGRID</td>
<td>64K</td>
<td>100MB</td>
</tr>
<tr>
<td>DrugBank</td>
<td>7K</td>
<td>60MB</td>
</tr>
<tr>
<td>Disease Ontology</td>
<td>10K</td>
<td>5MB</td>
</tr>
<tr>
<td>Protein Ontology</td>
<td>200K</td>
<td>130MB</td>
</tr>
<tr>
<td>Mesh Hierarchy</td>
<td>30K</td>
<td>40MB</td>
</tr>
<tr>
<td>PubChem</td>
<td>90M</td>
<td>1GB</td>
</tr>
<tr>
<td>DGIdb</td>
<td>5K</td>
<td>30MB</td>
</tr>
<tr>
<td>Gene Ontology</td>
<td>45K</td>
<td>10MB</td>
</tr>
<tr>
<td>MSigDB</td>
<td>14K</td>
<td>70MB</td>
</tr>
<tr>
<td>Reactome</td>
<td>20K</td>
<td>100MB</td>
</tr>
<tr>
<td>GEO</td>
<td>1.7M</td>
<td>80GB</td>
</tr>
<tr>
<td>ICGC (66 cancer projects)</td>
<td>40M</td>
<td>1TB</td>
</tr>
<tr>
<td>GTEx</td>
<td>50M</td>
<td>100GB</td>
</tr>
</tbody>
</table>

Total: 250M entities, 2.2TB raw data
Could all these datasets fit into RAM of a single machine?

Single machine prices:
- Server 1TB RAM, 80 cores, $25K
- Server 6TB RAM, 144 cores, $200K
- Server 12TB RAM, 288 cores, $400K

In my group we have 1TB RAM machines since 2012 and just got a 12TB RAM machine.
Dataset vs. RAM Sizes

- KD Nuggets survey since 2006 surveys: “What is the largest dataset you analyzed/mined?”
- Big RAM is eating big data:
  - Yearly increase of dataset sizes: 20%
  - Yearly increase of RAM sizes: 50%

Bottom line: Want to do graph analytics? Get a BIG machine!
## Trade-offs

<table>
<thead>
<tr>
<th>Option 1</th>
<th>Option 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard SQL database</td>
<td>Custom representations</td>
</tr>
<tr>
<td>Separate systems for tables and graphs</td>
<td>Integrated system for tables and graphs</td>
</tr>
<tr>
<td>Single representation for tables and graphs</td>
<td>Separate table and graph representations</td>
</tr>
<tr>
<td>Distributed system</td>
<td>Single machine system</td>
</tr>
<tr>
<td>Disk-based structures</td>
<td>In-memory structures</td>
</tr>
</tbody>
</table>

SNAP
Graph Analytics: SNAP

Unstructured data → Specify entities → Relational tables → Specify relationships → Tabular networks → Optimize representation → Network representation

Results → Integrate results → Perform graph analytics → SNAP
Experts on StackOverflow

posts

Select

Python questions

Join

Python answers

Python Q&A

ToGraph

SNAP

Q&A user network

HITS algorithm

Provenance Scripts

Scores

Jure Leskovec, Stanford
Graph Construction in SNAP

- SNAP (Python) code for executing finding the StackOverflow example

```python
P = ringo.LoadTable(schema,'posts.tsv')
JP = ringo.Select(P,'Tag=Java')
Q = ringo.Select(JP,'Type=question')
A = ringo.Select(JP,'Type=answer')
QA = ringoJoin(Q,A,'AnswerId','PostId')
G = ringo.ToGraph(QA,'UserId.1','UserId.2')
PR = ringo.GetPageRank(G)
S = ringo.ToTable(PR,'UserId','Score')
ringo.Save(S,'scores.bin')
```

SNAP Overview

High-Level Language User Front-End

- Interface with Graph Processing Engine
- Metadata (Provenance)
- Provenance Script

SNAP: In-memory Graph Processing Engine

- Filters
- Graph Methods
- Graph Containers
- Graph, Table Conversions
- Table Objects
- Secondary Storage
Input data must be manipulated and transformed into graphs.

<table>
<thead>
<tr>
<th>Src</th>
<th>Dst</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>v2</td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td>v3</td>
<td></td>
</tr>
<tr>
<td>v3</td>
<td>v4</td>
<td></td>
</tr>
<tr>
<td>v1</td>
<td>v3</td>
<td></td>
</tr>
<tr>
<td>v1</td>
<td>v4</td>
<td></td>
</tr>
</tbody>
</table>

Table data structure

Graph data structure
Creating a Graph in SNAP

Four ways to create a graph:

Nodes connected based on

(1) Pairwise node similarity
(2) Temporal order of nodes
(3) Grouping and aggregation of nodes
(4) The data already contains edges as source and destination pairs
Creating Graphs in SNAP (1)

Similarity-based: In a forum, connect users that post to similar topics

- Distance metrics
  - Euclidean, Haversine, Jaccard distance

- Connect similar nodes
  - SimJoin, connect if data points are closer than some threshold
  - How to get around quadratic complexity
    - Locality Sensitive Hashing
Sequence-based: In a Web log, connect pages in an order clicked by the users (click-trail)

- Connect a node with its $K$ successors
  - Events selected per user, ordered by timestamps
  - NextK, connect $K$ successors
Creating Graphs in SNAP (3)

- **Aggregation**: Measure the activity level of different user groups
  - **Edge creation**
    - Partition users to groups
    - Identify interactions within each group
    - Compute a score for each group based on interactions
  - Treat groups as super-nodes in a graph
Graphs and Methods

- SNAP supports several graph types
  - Directed, Undirected, Multigraph
- >200 graph algorithms
- Any algorithm works on any container
SNAP Implementation

- High-level front end
  - Python module
  - Uses SWIG for C++ interface
- High-performance graph engine
  - C++ based on SNAP
- Multi-core support
  - OpenMP to parallelize loops
  - Fast, concurrent hash table, vector operations
Graphs in SNAP

Nodes table

Sorted vectors of in- and out-neighbors

Nodes table

Sorted vectors of in- and out-edges

Edges table

Directed graphs in SNAP

Directed multigraphs in SNAP

Jure Leskovec, Stanford
## Experiments: Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>LiveJournal</th>
<th>Twitter2010</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>4.8M</td>
<td>42M</td>
</tr>
<tr>
<td>Edges</td>
<td>69M</td>
<td>1.5B</td>
</tr>
<tr>
<td>Text Size (disk)</td>
<td>1.1GB</td>
<td>26.2GB</td>
</tr>
<tr>
<td>Graph Size (RAM)</td>
<td>0.7GB</td>
<td>13.2GB</td>
</tr>
<tr>
<td>Table Size (RAM)</td>
<td>1.1GB</td>
<td>23.5GB</td>
</tr>
</tbody>
</table>
# Benchmarks, One Computer

<table>
<thead>
<tr>
<th>Algorithm Graph</th>
<th>PageRank LiveJournal</th>
<th>PageRank Twitter2010</th>
<th>Triangles LiveJournal</th>
<th>Triangles Twitter2010</th>
</tr>
</thead>
<tbody>
<tr>
<td>Giraph</td>
<td>45.6s</td>
<td>439.3s</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>GraphX</td>
<td>56.0s</td>
<td>-</td>
<td>67.6s</td>
<td>-</td>
</tr>
<tr>
<td>GraphChi</td>
<td>54.0s</td>
<td>595.3s</td>
<td>66.5s</td>
<td>-</td>
</tr>
<tr>
<td>PowerGraph</td>
<td>27.5s</td>
<td>251.7s</td>
<td>5.4s</td>
<td>706.8s</td>
</tr>
<tr>
<td>SNAP</td>
<td>2.6s</td>
<td>72.0s</td>
<td>13.7s</td>
<td>284.1s</td>
</tr>
</tbody>
</table>

Hardware: 4x Intel CPU, 64 cores, 1TB RAM, $35K

Jure Leskovec, Stanford
## Published Benchmarks

<table>
<thead>
<tr>
<th>System</th>
<th>Hosts</th>
<th>CPUs host</th>
<th>Host Configuration</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphChi</td>
<td>1</td>
<td>4</td>
<td>8x core AMD, 64GB RAM</td>
<td>158s</td>
</tr>
<tr>
<td>TurboGraph</td>
<td>1</td>
<td>1</td>
<td>6x core Intel, 12GB RAM</td>
<td>30s</td>
</tr>
<tr>
<td>Spark</td>
<td>50</td>
<td>2</td>
<td></td>
<td>97s</td>
</tr>
<tr>
<td>GraphX</td>
<td>16</td>
<td>1</td>
<td>8X core Intel, 68GB RAM</td>
<td>15s</td>
</tr>
<tr>
<td>PowerGraph</td>
<td>64</td>
<td>2</td>
<td>8x hyper Intel, 23GB RAM</td>
<td>3.6s</td>
</tr>
<tr>
<td><strong>SNAP</strong></td>
<td><strong>1</strong></td>
<td><strong>4</strong></td>
<td>20x hyper Intel, 1TB RAM</td>
<td><strong>6.0s</strong></td>
</tr>
</tbody>
</table>

Twitter2010, one iteration of PageRank
## SNAP: Sequential Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-core</td>
<td>31.0s</td>
</tr>
<tr>
<td>Single source shortest path</td>
<td>7.4s</td>
</tr>
<tr>
<td>Strongly connected components</td>
<td>18.0s</td>
</tr>
</tbody>
</table>

LiveJournal, 1 core
SNAP: Sequential Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time (s)</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-degree</td>
<td>14</td>
<td>1 core</td>
</tr>
<tr>
<td>Out-degree</td>
<td>8</td>
<td>1 core</td>
</tr>
<tr>
<td>PageRank</td>
<td>115</td>
<td>64 cores</td>
</tr>
<tr>
<td>Triangles</td>
<td>107</td>
<td>64 cores</td>
</tr>
<tr>
<td>WCC</td>
<td>1,716</td>
<td>1 core</td>
</tr>
<tr>
<td>K-core</td>
<td>2,325</td>
<td>1 core</td>
</tr>
</tbody>
</table>

- Benchmarks on the citation graph: nodes 50M, edges 757M
## SNAP: Tables and Graphs

<table>
<thead>
<tr>
<th>Dataset</th>
<th>LiveJournal</th>
<th>Twitter2010</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table to graph</td>
<td>8.5s</td>
<td>81.0s</td>
</tr>
<tr>
<td></td>
<td>13.0 MEdges/s</td>
<td>18.0 MEdges/s</td>
</tr>
<tr>
<td>Graph to table</td>
<td>1.5s</td>
<td>29.2s</td>
</tr>
<tr>
<td></td>
<td>46.0 MEdges/s</td>
<td>50.4 MEdges/s</td>
</tr>
</tbody>
</table>

Hardware: 4x Intel CPU, 80 cores, 1TB RAM, $35K
## SNAP: Table Operations

<table>
<thead>
<tr>
<th>Dataset</th>
<th>LiveJournal</th>
<th>Twitter2010</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select</td>
<td>&lt;0.1s</td>
<td>1.6s</td>
</tr>
<tr>
<td></td>
<td>575.0 MRows/s</td>
<td>917.7 MRows/s</td>
</tr>
<tr>
<td>Join</td>
<td>0.6s</td>
<td>4.2s</td>
</tr>
<tr>
<td></td>
<td>109.5 MRows/s</td>
<td>348.8 MRows/s</td>
</tr>
<tr>
<td>Load graph</td>
<td>5.2s</td>
<td>76.6s</td>
</tr>
<tr>
<td>Save graph</td>
<td>3.5s</td>
<td>69.0s</td>
</tr>
</tbody>
</table>

Hardware: 4x Intel CPU, 80 cores, 1TB RAM, $35K
Multimodal Networks:
A network of networks
Multimodal Networks

Mode

Node

In-mode links

Cross-mode links

Network of networks
Why multimodal networks?

- Can encode additional semantic structure than a “simple” graph
- Many naturally occurring graphs are multimodal networks
  - Gene-drug-disease networks
  - Social networks,
  - Academic citation graphs
Multimodal Network Example
Challenges

Multimoral network requirements:

- Fast processing
  - Efficient traversal of nodes and edges

- Dynamic structure
  - Quickly add/remove nodes and edges
    - Create subgraphs, dynamic graphs, ...

- Tradeoff
  - High performance, fixed structure
  - Highly flexible structure, low performance
Piggyback on a Graph

Why can’t we just piggyback extra information onto a regular graph?

- Want to ensure that per-mode information is easily accessible as a unit
- Want more fine-grained control as to where certain vertex and edge information resides
- Want indexes that allow for easy random access
Piggyback mode information

Benchmark multimodal graph:

Nodes in modes 0 to 9 are fully connected to each other

Each node in modes 0 to 9 is connected to 10 of the nodes in mode 10

- Modes 0 to 9 have 10K nodes each and 100M edges each
- Mode 10 has $X$ nodes
- Each node in modes 0 to 9 is connected to all nodes in mode 10

$X$ controls randomness of redundant edges (while the output size is fixed)

Jure Leskovec, Stanford
Experiment

For $X=1$M, graph has 10.1B edges

Extract subgraph on given modes

Workloads

- $SG(0,1)$
- $SG(0,1,4)$
- $SG(0$ to $9)$
- $G$NIds($0,1,3$)

Time (in seconds)

$x=1000$  $x=100000$  $x=1000000$

$x=10000$  $x=1000000$
How to be faster?

- **Remember:** Everything is in memory so don’t need to worry about disk

- **Desirable properties:**
  - Stay in cache as much as possible as memory accesses are expensive in comparison
    - i.e., we want good memory locality
  - Cheap index lookups that allow us to avoid having to look at the entire data structure
Multimodal Networks

- **Idea 1:** Represent the multimodal graph as a collection of bipartite graphs
- **Idea 2:** Consolidate node hash tables
- **Idea 3:** Consolidate adjacency lists
Idea 1: BGC

BGC (Bipartite Graph Collection): Collection of per-mode bipartite graphs

Nodes can be repeated across different graphs

k(k+1)/2 bipartite graphs, each bipartite graph has its own node hash table

Each node object in a node hash table maps to a list of in- and out-neighbors
Idea 2: Hybrid

**Hybrid**: Collection of per-mode node hash tables along with individual per-mode adjacency lists

- **Nodes only appear in a single node hash table**
- **k node hash tables**
- Each node object in a node hash table maps to $k$ lists of in- and out-neighbors sorted by node-id
Idea 3: MNCA

MNCA (Multi-node hash table, consolidated adjacency lists): Per-mode node hash tables + big adjacency list

Nodes only appear in a single node hash table

\( k \) node hash tables

Each node object in a node hash table maps to a consolidated list of in- and out-neighbors sorted by (mode-id, node-id)
So, how do we do?

3.5x order of magnitude improvement!

11 modes in total
- 10k nodes in modes 0-9; edges between all nodes
- 1M nodes in mode 10; edges between every node in mode 10 and all other nodes (total of 110B edges)
Tradeoffs by Workload

- Workload type:

<table>
<thead>
<tr>
<th>BGC</th>
<th>Hybrid</th>
<th>MNCA</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>✔️</td>
<td>✔</td>
</tr>
<tr>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
</tbody>
</table>

  - Per-mode NodeId lookups
  - All-adjacent NodeId accesses
  - Per-mode adjacent NodeId accesses
  - Mode-pair SubGraph accesses
Tradeoffs by Graph Type

- **Graph type:**

  - **BGC**
  - **Hybrid**
  - **MNCA**

  - Sparser graphs
  - Denser graphs
  - Number of out-neighbors
Latest Algorithms: Feature Learning in Graphs

node2vec: Scalable Feature Learning for Networks
(Supervised) Machine Learning Lifecycle: This feature, that feature. Every single time!
Feature Learning in Graphs

Goal: Learn features for a set of objects

Feature learning in graphs:
- **Given:** $G = (V, E)$
- **Learn a function:** $f : V \rightarrow \mathbb{R}^d$
  - **Not task specific:** Just given a graph, learn $f$. Can use the features for any downstream task!
Unsupervised Feature Learning

- **Intuition:** Find a mapping of nodes to $d$-dimensions that preserves some sort of node similarity

- **Idea:** Learn node embedding such that nearby nodes are close together

- **Given a node $u$, how do we define nearby nodes?**
  - $N_S(u) \ldots$ neighbourhood of $u$ obtained by sampling strategy $S$
Unsupervised Feature Learning

- **Goal**: Find embedding that predicts nearby nodes $N_S(u)$:

\[
\max_f \sum_{u \in V} \log Pr(N_S(u) | f(u))
\]

- **Make independence assumption**:

\[
Pr(N_S(u) | f(u)) = \prod_{n_i \in N_S(u)} Pr(n_i | f(u))
\]

\[
Pr(n_i | f(u)) = \frac{\exp(f(n_i) \cdot f(u))}{\sum_{v \in V} \exp(f(v) \cdot f(u))}
\]

Estimate $f(u)$ using stochastic gradient descent.
How to determine $N_S(u)$

Two classic search strategies to define a neighborhood of a given node:

1. BFS: Breadth-First Search
2. DFS: Depth-First Search

For $|N_S(u)| = 3$

$u = \{s_1, s_2, s_3\}$
BFS vs. DFS

BFS: Micro-view of neighbourhood

DFS: Macro-view of neighbourhood

Structural vs. Homophilic equivalence
Parameters were learned using 10-fold cross-validation on just 10% of the data. Following prior work [34], we use the same hyperparameters used for DeepWalk and LINE. Specifically, for DeepWalk, we let it run for all benchmark datasets. As an example, if we consider DeepWalk, it has a parameter \( k \) for specifying the minimum number of context neighbors to optimize for and the greater the number, the more nodes are used. DeepWalk have a parameter \( d \) for approximation in the sampling phase, \( k \) for specifying the number of neighbors to be considered, and \( q \) for the number of random walks to be generated. In the optimization phase, all benchmarks as well as node2vec approaches, we let it run for all benchmark datasets.

As an example, if we consider DeepWalk, it has a parameter \( k \) for specifying the minimum number of context neighbors to optimize for and the greater the number, the more nodes are used. DeepWalk have a parameter \( d \) for approximation in the sampling phase, \( k \) for specifying the number of neighbors to be considered, and \( q \) for the number of random walks to be generated. In the optimization phase, all benchmarks as well as node2vec approaches, we let it run for all benchmark datasets.

In the sampling phase, all benchmarks as well as node2vec approaches, we let it run for all benchmark datasets. As an example, if we consider DeepWalk, it has a parameter \( k \) for specifying the minimum number of context neighbors to optimize for and the greater the number, the more nodes are used. DeepWalk have a parameter \( d \) for approximation in the sampling phase, \( k \) for specifying the number of neighbors to be considered, and \( q \) for the number of random walks to be generated. In the optimization phase, all benchmarks as well as node2vec approaches, we let it run for all benchmark datasets.

In the multi-label classification setting, every node is assigned with label colors. The task is to predict the labels of the remaining nodes. During the training phase, we perform multi-label classification with label colors. The learned node feature representations are input to a one-vs-rest logistic regression using the LIBLINEAR form functions complementary to those of neighboring proteins, which respond to the graph induced by nodes. The subgraphs are evaluated in settings that equalize for runtime. In doing so, we note that the runtime complexity is contributed from two distinct implementations: language (C/C++/Python) since it is secondary to the fact that all other comparison benchmarks are evaluated in settings that equalize for runtime. In doing so, we note that the runtime complexity is contributed from two distinct implementations: language (C/C++/Python) since it is secondary to the fact that all other comparison benchmarks are evaluated in settings that equalize for runtime. In doing so, we note that the runtime complexity is contributed from two distinct implementations: language (C/C++/Python) since it is secondary to the fact that all other comparison benchmarks are evaluated in settings that equalize for runtime.

For comparing performance in Table 2, we use a subgraph for the protein-protein interactions (PPI) network for Homo Sapiens. The subgraph corresponds to the graph induced by nodes for which we could respond to the graph induced by nodes for which we could. The learned node feature representations are input to a one-vs-rest logistic regression using the LIBLINEAR form functions complementary to those of neighboring proteins, which respond to the graph induced by nodes. The subgraphs are evaluated in settings that equalize for runtime. In doing so, we note that the runtime complexity is contributed from two distinct implementations: language (C/C++/Python) since it is secondary to the fact that all other comparison benchmarks are evaluated in settings that equalize for runtime.
Interpolating BFS and DFS

- Biased random walk procedure, that given a node $u$ samples $N_s(u)$

The walk just traversed $(t, v)$ and aims to make a next step.
### Multilabel Classification

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Spectral Clustering</td>
<td>0.0405</td>
<td>0.0681</td>
<td>0.0395</td>
</tr>
<tr>
<td>DeepWalk</td>
<td>0.2110</td>
<td>0.1768</td>
<td>0.1274</td>
</tr>
<tr>
<td>LINE</td>
<td>0.0784</td>
<td>0.1447</td>
<td>0.1164</td>
</tr>
<tr>
<td>node2vec</td>
<td><strong>0.2581</strong></td>
<td><strong>0.1791</strong></td>
<td><strong>0.1552</strong></td>
</tr>
</tbody>
</table>

*node2vec settings (p,q)*: 0.25, 0.25  
*Gain of node2vec [%]*: **22.3**  

- Spectral embedding
- DeepWalk [B. Perozzi et al., KDD ‘14]
- LINE [J. Tang et al., WWW ‘15]
Incomplete Network Data (PPI)

![Graphs showing the relationship between Macro-F1 score and the fraction of missing edges and additional edges.](image-url)
Conclusion
Conclusion

- Big-memory machines are here:
  - 1TB RAM, 100 Cores ≈ a small cluster
  - No overheads of distributed systems
  - Easy to program

- Most “useful” datasets fit in memory

- Big-memory machines present a viable solution for analysis of all-but-the-largest networks
Graphs have to be Built

- Graphs have to be built from data
  - Processing of tables and graphs

Jure Leskovec, Stanford
Multimodal Networks

- Graphs are more than wiring diagrams
- Multimodal network: A network of Networks
- Building scalable data structures
- NUMA architectures provide interesting new tradeoffs

Jure Leskovec, Stanford
Building Robust Systems

How to get robust performance always?

- Ongoing/future work
  - Better characterize the optimal representation required given workload and graph type
  - Try to dynamically switch representations when nodes get sufficiently high degrees or particular queries become more common
  - Benchmark on real data and real queries
References

- **Papers:**
  - **SNAP: A General Purpose Network Analysis and Graph Mining Library.**

- **Software:**
  - [http://snap.stanford.edu/snappy](http://snap.stanford.edu/snappy)
  - [https://github.com/snap-stanford/snap](https://github.com/snap-stanford/snap)
THANKS!

@jure
http://snap.stanford.edu