Subsampling Graphs
RECAP OF PAGERANK-NIBBLE
Why I’m talking about graphs

• Lots of large data is graphs
  – Facebook, Twitter, citation data, and other social networks
  – The web, the blogosphere, the semantic web, Freebase, Wikipedia, Twitter, and other information networks
  – Text corpora (like RCV1), large datasets with discrete feature values, and other bipartite networks
    • nodes = documents or words
    • links connect document → word or word → document
  – Computer networks, biological networks (proteins, ecosystems, brains, ...), ...
  – Heterogeneous networks with multiple types of nodes
    • people, groups, documents
Our first operation on graphs

• Local graph partitioning
• Why?
  – it’s about the best we can do in terms of subsampling/exploratory analysis of a graph
What is Local Graph Partitioning?

A local graph partitioning algorithm finds a small cut near the given seed(s) with running time depending only on the size of the output.
What is Local Graph Partitioning?

Submarkets in the bigging graph

The bidding graph has numerous submarkets, related to real estate, flower delivery, hotels, gambling, ...

It is useful to identify these submarkets.

- Find groups of related phrases to suggest to advertisers.
- Find small submarkets for testing and experimentation.
Main results of the paper (Reid et al)

1. An *approximate* personalized PageRank computation that only touches nodes “near” the seed
   – but has small error relative to the true PageRank vector
2. A proof that a *sweep* over the approximate personalized PageRank vector finds a cut with conductance $\sqrt{\alpha \ln m}$
   – unless no good cut exists
   • no subset $S$ contains significantly more mass in the approximate PageRank than in a uniform distribution
Approximate PageRank: Key Idea

\[ pr(\alpha, s) = \alpha s + (1 - \alpha)pr(\alpha, sW). \quad W = \frac{I+P}{2} \]

\[ \text{push}_u(p, r): \]

1. Let \( p' = p \) and \( r' = r \), except for the following changes:
   
   (a) \( p'(u) = p(u) + \alpha r(u) \).
   
   (b) \( r'(u) = (1 - \alpha)r(u)/2 \).
   
   (c) For each \( v \) such that \( (u, v) \in E \) : \( r'(v) = r(v) + (1 - \alpha)r(u)/(2d(u)) \).

2. Return \( (p', r') \).

- \( p \) is current approximation (start at 0)
- \( r \) is set of “recursive calls to make”
  
  - residual error
  
  - start with all mass on \( s \)
- \( u \) is the node picked for the next call
ANALYSIS OF PAGERANK-NIBBLE
Analysis

Lemma 1. Let $p'$ and $r'$ be the result of the operation $\text{push}_u$ on $p$ and $r$. Then

$$p' + \text{pr}(\alpha, r') = p + \text{pr}(\alpha, r).$$

Proof of Lemma 1. After the push operation, we have

$$p' = p + \alpha r(u) \chi_u,$$
$$r' = r - r(u) \chi_u + (1 - \alpha) r(u) \chi_u W.$$

Using equation (5),

$$p + \text{pr}(\alpha, r) = p + \text{pr}(\alpha, r - r(u) \chi_u) + \text{pr}(\alpha, r(u) \chi_u)$$
$$= p + \text{pr}(\alpha, r - r(u) \chi_u) + [\alpha r(u) \chi_u + (1 - \alpha) \text{pr}(\alpha, r(u) \chi_u W)]$$
$$= [p + \alpha r(u) \chi_u] + \text{pr}(\alpha, [r - r(u) \chi_u + (1 - \alpha) r(u) \chi_u W])$$
$$= p' + \text{pr}(\alpha, r').$$

$$\text{pr}(\alpha, s) = \alpha s + (1 - \alpha) \text{pr}(\alpha, sW).$$
Approximate PageRank: Key Idea

By definition PageRank is fixed point of:

\[ \text{pr}(\alpha, s) = \alpha s + (1 - \alpha) \text{pr}(\alpha, s) W, \]

Claim:

\[ \text{pr}(\alpha, s) = \alpha s + (1 - \alpha) \text{pr}(\alpha, sW). \]

Proof:

\[ \begin{align*}
R_\alpha &= \alpha \sum_{t=0}^{\infty} (1 - \alpha)^t W^t \\
&= \alpha I + (1 - \alpha) W R_\alpha.
\end{align*} \]

\[ \begin{align*}
\text{pr}(\alpha, s) &= s R_\alpha \\
&= \alpha s + (1 - \alpha) s W R_\alpha \\
&= \alpha s + (1 - \alpha) \text{pr}(\alpha, sW). \]
**Approximate PageRank: Algorithm**

\[ \text{ApproximatePageRank} (v, \alpha, \epsilon): \]

1. Let \( p = 0 \), and \( r = \chi_v \).

2. While \( \max_{u \in V} \frac{r(u)}{d(u)} \geq \epsilon \):
   (a) Choose any vertex \( u \) where \( \frac{r(u)}{d(u)} \geq \epsilon \).
   (b) Apply \( \text{push}_u \) at vertex \( u \), updating \( p \) and \( r \).

3. Return \( p \), which satisfies \( p = \text{apr}(\alpha, \chi_v, r) \) with \( \max_{u \in V} \frac{r(u)}{d(u)} < \epsilon \).

\[ \text{push}_u(p, r): \]

1. Let \( p' = p \) and \( r' = r \), except for the following changes:
   (a) \( p'(u) = p(u) + \alpha r(u) \).
   (b) \( r'(u) = (1 - \alpha)r(u)/2 \).
   (c) For each \( v \) such that \( (u, v) \in E \):
      \( r'(v) = r(v) + (1 - \alpha)r(u)/(2d(u)) \).

2. Return \( (p', r') \).
Analysis

Lemma 1. Let $p'$ and $r'$ be the result of the operation $\text{push}_u$ on $p$ and $r$. Then

$$p' + \text{pr}(\alpha, r') = p + \text{pr}(\alpha, r).$$

So, at every point in the apr algorithm:

$$p + \text{pr}(\alpha, r) = \text{pr}(\alpha, \chi_v),$$

Also, at each point, $|r|_1$ decreases by $\alpha * \varepsilon * \text{degree}(u)$, so:

after $T$ push operations where $\text{degree}(i\text{-th } u)=d_i$, we know

$$\sum_{i} d_i \cdot \alpha \varepsilon \leq 1 \quad \Rightarrow \quad \sum_{i=1}^{T} d_i \leq \frac{1}{\varepsilon \alpha}.$$
Theorem 1. ApproximatePageRank\((v, \alpha, \epsilon)\) runs in time \(O(\frac{1}{\epsilon \alpha})\), and computes an approximate PageRank vector \(p = \text{apr}(\alpha, \chi_v, r)\) such that the residual vector \(r\) satisfies \(\max_{u \in V} \frac{r(u)}{d(u)} < \epsilon\), and such that \(\text{vol}(\text{Supp}(p)) \leq \frac{1}{\epsilon \alpha}\).

With the invariant: \[ p + \text{pr}(\alpha, r) = \text{pr}(\alpha, \chi_v), \]

This bounds the error of \(p\) relative to the PageRank vector.
ApproximatePageRank \((v, \alpha, \epsilon)\):

1. Let \(p = \vec{0}\), and \(r = \chi_v\).
2. While \(\max_{u \in V} \frac{r(u)}{d(u)} \geq \epsilon\):
   
   (a) Choose any vertex \(u\) where \(\frac{r(u)}{d(u)} \geq \epsilon\).
   
   (b) Apply \(\text{push}_u\) at vertex \(u\), updating \(p\) and \(r\).
3. Return \(p\), which satisfies \(p = \text{apr}(\alpha, \chi_v, r)\) with \(\max_{u \in V} \frac{r(u)}{d(u)} < \epsilon\).

\[
d(v) = \text{api}.\text{degree}(v)
\]

\(p, r\) are hash tables – they are small \((1/\epsilon \alpha)\)

Could implement with API:
- List<
  Node\>>\ neighbor(Node \(u\))
- int degree(Node \(u\))

push\(_u\)(\(p, r\)):

1. Let \(p' = p\) and \(r' = r\), except for the following changes:

   (a) \(p'(u) = p(u) + \alpha r(u)\).
   
   (b) \(r'(u) = (1 - \alpha)r(u)/2\).
   
   (c) For each \(v\) such that \((u, v) \in E\):
       \(r'(v) = r(v) + (1 - \alpha)r(u)/(2d(u))\).
2. Return \((p', r')\).
Comments - Ordering

ApproximatePageRank \((v, \alpha, \epsilon)\):

1. Let \(p = \vec{0}\), and \(r = \chi_v\).

2. While \(\max_{u \in V} \frac{r(u)}{d(u)} \geq \epsilon\):
   
   (a) Choose any vertex \(u\) where \(\frac{r(u)}{d(u)} \geq \epsilon\).
   
   (b) Apply \texttt{push}_u\) at vertex \(u\), updating \(p\) and \(r\).

3. Return \(p\), which satisfies \(p = \text{apr}(\alpha, \chi_v, r)\) with \(\max_{u \in V} \frac{r(u)}{d(u)} < \epsilon\).

\texttt{push}_u(p, r):

1. Let \(p' = p\) and \(r' = r\), except for the following changes:

   (a) \(p'(u) = p(u) + \alpha r(u)\).

   (b) \(r'(u) = (1 - \alpha)r(u)/2\).

   (c) For each \(v\) such that \((u, v) \in E\): \(r'(v) = r(v) + (1 - \alpha)r(u)/(2d(u))\).

2. Return \((p', r')\).
ApproximatePageRank \((v, \alpha, \epsilon)\):

1. Let \(p = \vec{0}\), and \(r = \chi_v\).
2. While \(\max_{u \in V} \frac{r(u)}{d(u)} \geq \epsilon\):

   Scan \textbf{repeatedly} through an adjacency-list encoding of the graph

   For every line you read \(u, v_1, \ldots, v_{d(u)}\) such that \(r(u)/d(u) > \epsilon\):

   \[(b) \text{ Apply push}_u \text{ at vertex } u, \text{ updating } p \text{ and } r.\]

3. Return \(p\), which satisfies \(p = \text{apr}(\alpha, \chi_v, r)\) with \(\max_{u \in V} \frac{r(u)}{d(u)} < \epsilon\).

\textbf{benefit:} storage is \(O(1/\epsilon \alpha)\) for the hash tables, avoids any \textit{seeking}
Possible optimizations?

• Much faster than doing random access the first few scans, but then slower the last few
  • ...there will be only a few ‘pushes’ per scan
• Optimizations you might imagine:
  – Parallelize?
  – Hybrid seek/scan:
    • Index the nodes in the graph on the first scan
    • Start seeking when you expect too few pushes to justify a scan
      – Say, less than one push/megabyte of scanning
  – Hotspots:
    • Save adjacency-list representation for nodes with a large $r(u)/d(u)$ in a separate file of “hot spots” as you scan
    • Then rescan that smaller list of “hot spots” until their score drops below threshold.
After computing apr…

• Given a graph
  – that’s too big for memory, and/or
  – that’s only accessible via API
• ...we can extract a sample in an interesting area
  – Run the apr/rwr from a seed node
  – Sweep to find a low-conductance subset
• Then
  – compute statistics
  – test out some ideas
  – visualize it...
Key idea: a “sweep”

• Order vertices $v_1, v_2, \ldots$ by highest $apr(s)$
• Pick a prefix $S = \{ v_1, v_2, \ldots, v_k \}$:
  – $S = \{ \}; \ volS = 0; B = \{ \}$
  – For $k = 1, \ldots, n$:
    • $S += \{ v_k \}$
    • $volS += degree(v_k)$
    • $B += \{ u: v_k \rightarrow u \text{ and } u \text{ not in } S \}$
    • $\Phi[k] = |B|/volS$
  – Pick $k$: $\Phi[k]$ is smallest
Scoring a graph prefix

\[ \partial(S) = \{ \{x, y\} \in E \mid x \in S, y \notin S \} \]

the edges leaving S

\[ \Phi(S) = \frac{|\partial(S)|}{\min (\text{vol}(S), 2m - \text{vol}(S))}. \]

- vol(S) is sum of deg(x) for x in S
- for small S: Prob(random edge leaves S)
Putting this together

• Given a graph
  – that’s too big for memory, and/or
  – that’s only accessible via API
• ...we can extract a sample in an interesting area
  – Run the apr/rwr from a seed node
  – Sweep to find a low-conductance subset
• Then
  – compute statistics
  – test out some ideas
  – visualize it...
Visualizing a Graph with Gephi
Screen shots/demo

- Gelphi – java tool
- Reads several inputs
  - .csv file
  - [demo]
### Role-playing_games – Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>10.0</td>
</tr>
<tr>
<td>Position (x)</td>
<td>-284.15045</td>
</tr>
<tr>
<td>Position (y)</td>
<td>-2252.2498</td>
</tr>
<tr>
<td>Position (z)</td>
<td>0.0</td>
</tr>
<tr>
<td>Color</td>
<td>[153,153,153]</td>
</tr>
</tbody>
</table>

### Role-playing_games – Attributes

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Role-playing_games</td>
</tr>
<tr>
<td>Label</td>
<td>Role-playing_games</td>
</tr>
</tbody>
</table>
PageRank
Ranks nodes "pages" according to how often a user following links will non-randomly reach the node "page".

- Directed
  - Probability (p): 0.85
  - Used to simulate the user randomly restarting the web-surfing.
- Undirected
  - Epsilon: 0.0010
  - Stopping criterion, the smaller this value, the longer convergence will take.

Use edge weight □

Cancel  OK
<table>
<thead>
<tr>
<th>Nodes</th>
<th>Id</th>
<th>Label</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Living_people</td>
<td>Living_people</td>
<td>Living_people</td>
<td>0.006</td>
</tr>
<tr>
<td>Mathematics</td>
<td>Mathematics</td>
<td>Mathematics</td>
<td>0.004</td>
</tr>
<tr>
<td>Computer_science</td>
<td>Computer_science</td>
<td>Computer_science</td>
<td>0.004</td>
</tr>
<tr>
<td>Statistics</td>
<td>Statistics</td>
<td>Statistics</td>
<td>0.003</td>
</tr>
<tr>
<td>Java_programming_language</td>
<td>Java_programming_language</td>
<td>Java_programming_language</td>
<td>0.003</td>
</tr>
<tr>
<td>C_programming_language</td>
<td>C_programming_language</td>
<td>C_programming_language</td>
<td>0.003</td>
</tr>
<tr>
<td>Programming_language</td>
<td>Programming_language</td>
<td>Programming_language</td>
<td>0.003</td>
</tr>
<tr>
<td>Compiler</td>
<td>Compiler</td>
<td>Compiler</td>
<td>0.003</td>
</tr>
<tr>
<td>Object-oriented_programming</td>
<td>Object-oriented_programming</td>
<td>Object-oriented_programming</td>
<td>0.003</td>
</tr>
<tr>
<td>Year_of_birth_mising</td>
<td>Year_of_birth_mising</td>
<td>Year_of_birth_mising</td>
<td>0.002</td>
</tr>
<tr>
<td>Probability_theory</td>
<td>Probability_theory</td>
<td>Probability_theory</td>
<td>0.002</td>
</tr>
<tr>
<td>Real_number</td>
<td>Real_number</td>
<td>Real_number</td>
<td>0.002</td>
</tr>
<tr>
<td>Set_mathematics</td>
<td>Set_mathematics</td>
<td>Set_mathematics</td>
<td>0.002</td>
</tr>
<tr>
<td>Data_type</td>
<td>Data_type</td>
<td>Data_type</td>
<td>0.002</td>
</tr>
<tr>
<td>Computer</td>
<td>Computer</td>
<td>Computer</td>
<td>0.001</td>
</tr>
<tr>
<td>Computer_programming</td>
<td>Computer_programming</td>
<td>Computer_programming</td>
<td>0.001</td>
</tr>
<tr>
<td>Random_variable</td>
<td>Random_variable</td>
<td>Random_variable</td>
<td>0.001</td>
</tr>
</tbody>
</table>
---Choose a rank parameter---

Hierarchy | Graph

Mouse selection

C_28programming_language29 - Properties

Size 27.594063
Position (x) -193.48184
Position (y) 880.3036
Position (z) 0.0
Color [153,153,153]

C_28programming_language29 - Attributes

Id C_28programming_language29
Label C_28programming_language29
PageRank 0.002981598053353604

C_28programming_language29

ForceAtlas 2

Run
Network Overview
- Average Degree
- Avg. Weighted Degree
- Network Diameter
- Graph Density
- HITS
- Modularity
- PageRank
- Connected Components

Node Overview
- Avg. Clustering Coefficient
- Eigenvector Centrality

Edge Overview
- Avg. Path Length