Graph Algorithms

- Recall basic Graph definition:
  - A graph is a pair (of sets) \( G = (V,E) \)
  - \( V \) is a finite set, called the “vertices”
  - \( E \) is a set of pairs of elements from \( V \), denoting the “edges” of the graph
Review of types of graphs

- Weighted: comes with a function $w: E \rightarrow \mathbb{R}$
  - Denotes weight on each edge
- Simple: no self-loop edges
- Directed: edges are one-way
- Acyclic: no cycles
- Complete: all possible edges
- Connected: path between every pair of vertices
Important Graph Problems

- Minimal spanning tree
  - (in a weighted graph): find the smallest spanning tree, in terms of total edge weight

- Calculate shortest paths between nodes
  - Either weighted or unweighted

- Maximal independent set
  - Largest set of vertices that have no common edges
Prim’s Algorithm

- Idea: start from a single node, then iteratively and “greedily” add vertices to the MST until all are added

\[
\text{MST-PRIM}(G, w, r) \\
\text{1 for each } u \in G.V \\
\text{2 } u.\text{key} = \infty \\
\text{3 } u.\pi = \text{NIL} \\
\text{4 } r.\text{key} = 0 \\
\text{5 } Q = G.V \\
\text{6 while } Q \neq \emptyset \\
\text{7 } u = \text{EXTRACT-MIN}(Q) \\
\text{8 for each } v \in G.\text{Adj}[u] \\
\text{9 \quad if } v \in Q \text{ and } w(u, v) < v.\text{key} \\
\text{10\quad } v.\pi = u \\
\text{11\quad } v.\text{key} = w(u, v)
\]
Parallelizing Prim’s Alg.

- Distribute columns or rows of the adjacency matrix
- The inner-loop can be re-framed as a reduce operation, using the MAX operation
- The structure Q can be a priority queue for efficiency, also distributed.
  - Then the selection of the min can be a reduction as well
  - Can keep a copy of the full queue on each node for easier bookkeeping
Parallel Prim’s Algorithm

- Cost to select the minimum entry
  - $O(n/p)$: scan $n/p$ local part of vector on processors
  - $O(\log p)$ all-to-one reduction across processors
- Broadcast next node selected for membership
  - $O(\log p)$
- Cost of locally updating $d$ vector
  - $O(n/p)$: replace $d$ with min of $d$ and matrix row
- Parallel time per iteration
  - $O(n/p + \log p)$
- Total parallel time
  - $O(n^2/p + n \log p)$
Single-Start Shortest Path

- Use BFS!
  - The “d” property after completion is the length of the s.p.
  - The “pi” attribute is the predecessor in that s.p.
- Not very easy to make parallel :(  

```plaintext
BFS(G, s)
1   for each vertex u ∈ G.V − {s}
2       u.color = WHITE
3       u.d = ∞
4       u.π = NIL
5   s.color = GRAY
6   s.d = 0
7   s.π = NIL
8   Q = ∅
9   ENQUEUE(Q, s)
10  while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13       if v.color == WHITE
14         v.color = GRAY
15         v.d = u.d + 1
16         v.π = u
17         ENQUEUE(Q, v)
18     u.color = BLACK
```
All-Pairs Shortest Path

- Do them all at once!
- Floyd-Warshall Algorithm:
  - Use dynamic programming
  - The length of the min path between nodes has a “nice” recursive structure
  - Idea: iterate \( k = 0 \) to \(|V|\) times, each time increasing the length of all paths

\[
d_{ij}^{(k)} = \begin{cases} 
  w_{ij} & \text{if } k = 0, \\
  \min \left( d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) & \text{if } k \geq 1 .
\end{cases}
\]
Floyd-Warshall Algorithm

FLOYD-WARSHALL (W)
1  \( n = W.\text{rows} \)
2  \( D^{(0)} = W \)
3  \( \text{for } k = 1 \text{ to } n \)
4  \( \quad \text{let } D^{(k)} = (d_{ij}^{(k)}) \text{ be a new } n \times n \text{ matrix} \)
5  \( \quad \text{for } i = 1 \text{ to } n \)
6  \( \quad \quad \text{for } j = 1 \text{ to } n \)
7  \( \quad \quad \quad d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \)
8  \( \quad \text{return } D^{(n)} \)

- We can “save” the paths in a matrix as we calculate the distance updates:

\[
\pi_{ij}^{(k)} = \begin{cases} 
\pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\
\pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. 
\end{cases}
\]
Parallel Floyd-Warshall

• Split the Adjacency matrix block-wise and parallelize the two inner loops:

```
procedure FLOYD_2DBLOCK(D(0))
begin
  for k := 1 to n do
  begin
    each process \( P_{i,j} \) that has a segment of the \( k^{th} \) row of \( D^{(k-1)} \);
    broadcasts it to the \( P_{*,j} \) processes;
    each process \( P_{i,j} \) that has a segment of the \( k^{th} \) column of \( D^{(k-1)} \);
    broadcasts it to the \( P_{i,*} \) processes;
    each process waits to receive the needed segments;
    each process \( P_{i,j} \) computes its part of the \( D^{(k)} \) matrix;
  end
end FLOYD_2DBLOCK
```
Parallel Floyd-Warshall

\[(i-1) \cdot \frac{n}{\sqrt{p}} + 1, (j-1) \cdot \frac{n}{\sqrt{p}} + 1\]

\[d_{k,r}^{(k-1)}, d_{l,k}^{(k-1)}\]

\[d_{l,r}^{(k)}\]
Independent Sets

• Want to find the largest independent set of vertices (no edges between)

• Simple naive algorithm:
  – Start with an empty set of vertices
  – Add vertex with smallest degree; remove its neighbors from G
    • Repeat this until G has no vertices left
• Very difficult to parallize!
  – As in, can’t be done :(}
Luby’s Max Ind. Set

• Randomized!

• Algorithm:
  – Start with an empty set
  – Assign random numbers to each vertex
  – Add vertices that got a number smaller than all neighbors to the set, remove nbrs
  – Repeat above two steps until G empty

• Good to parallelize?
  – Yes!
Strategies for Graphs

• Spoiler: use linear algebra!
  – Easy to distribute
  – Lets us leverage data reuse and cache locality

• Use the adjacency matrix of the graph. E.g:
  – If we have vector of distances from a single vertex (e.g. after BFS), we can re-formulate the kinds of loops above
  – Define a custom “inner product”:

\[ y^t = x^t G \equiv \forall i: y_j = \min \{ x_j, \min_{i: G_{ij} \neq 0} \{ x_i + 1 \} \} \]
Example: Bellman-Ford

- Finds shortest path from a single source, allowing negative edge weights

  Let \( s \) be given, and set \( d(s) = 0 \)
  Set \( d(v) = \infty \) for all other nodes \( v \)
  for \( |E| - 1 \) times do
    for all edges \( e = (u, v) \) do
      Relax: if \( d(u) + w_{uv} < d(v) \) then
        Set \( d(v) \leftarrow d(u) + w_{uv} \)

- Easily parallelized now, using the above vector-matrix product!
Example: Search Engine

- Model web-pages with hyperlinks between them as a graph:

\[ L_{ij} = \begin{cases} 
  1 & \text{document } i \text{ points to document } j \\
  0 & \text{otherwise} 
\end{cases} \]

- Every webpage will have an “authority” and “hub” score

- If we say \( x \) is the vector of authorities and \( y \) the vector of hub scores:

\[
\begin{align*}
  x &= L^t y \\
  y &= Lx
\end{align*}
\]
HITS (hypertext-induced search)

• Using the previous formulation (after some substitution), we get $x = LL^tx$ and $y = L^tLy$

• In other words, this is an eigenvalue problem!
  – Saved by linear algebra again!

• How do we compute these then?
Eigenvalue Problems

<preach>
Some of the most important types of problems in practical scenarios. Period.
</preach>

- Most of “data analysis” is really just solving fancy eigenvalue problems
- Eigenvalues capture “importance” of data, as just mentioned earlier in the context of web-pages
Eigenvalue Problems

- Eigenvalues/Eigenvectors:
  - Given a matrix $A$, we say that $x$ is an eigenvector with corresponding (scalar) eigenvalue $t$ if
    $$Ax = tx$$
  - i.e. Applying $A$ to $x$ only stretches the vector
  - i.e. $x$ lies along one of the important “directions” of the matrix $A$

- Matrices typically have many eigenvectors
Eigenvalues/Eigenvectors
Solving Eigenvalue Systems

- Given a matrix $A$
- Choose a vector (hopefully smartly) $x_0$
- Perform the sequential update: $x_i = A x_{i-1}$
- This is called the power method because we end up with $x_i = A^i x_0$.
- If we got lucky with starting point, we would simply have $A x_0 = \lambda x_0$ and $x_i = \lambda^i x_0$. 
Example

• Try the following matrix $A$ and starting vector $x$:

$$A = \begin{pmatrix}
1 & 1 & 1 \\
1 & \ddots & 1 \\
1 & 1 & 1
\end{pmatrix}, \quad x = (0, \ldots, 0, 1)^t.$$

• Try by hand for small examples. What happens?
Back to HITS...

Input: Adjacency matrix $A$ of size $n \times m$ and number of iterations
Output: Authority and hub score vectors $x$ and $y$ respectively
$x = (1, 1, \ldots, 1) \in \mathcal{R}^m$; $y = (1, 1, \ldots, 1) \in \mathcal{R}^n$;
while Iterations still left do  // Can detect when $x$ & $y$ don’t change anymore
    for $i=1,2,\ldots,m$ do
        $x_j = \sum_{a_{ij}=1} y_i$;  // This is a vector-matrix product!
    end
    for $j=1,2,\ldots,n$ do
        $y_i = \sum_{a_{ij}=1} x_j$;  // This is a vector-matrix product!
    end
    Normalize($x$); Normalize($y$);
end

• Note: normalization means adjust to have unit length (root of sum of squares)
PageRank

• Start with vector \( p = e = \text{vector of all 1's} \)
• Use the adjacency matrix \( M \) where \( M_{ij} = 1 \) if page \( j \) has a link to page \( i \)
  - Then normalize it so that columns have a sum of 1, making it a stochastic matrix
• Choose small constant \( s > 0 \) to represent the chance of moving to a page that is not linked
• Iterate the process:
  - \( p = s \cdot M \cdot p + s \cdot e \)
PageRank

- Assuming the above process converges, what can we say analytically about the situation?
- Fixed point is: $p = s \cdot M \cdot p + s \cdot e$
- Then $(I_n - sM) \cdot p = s \cdot e$
- So what we really need is $(I_n - sM)$ to have an inverse
- But if it does, it must be of the form
  - $I_n + sM + s^2M^2 + s^2M^2 \ldots$
  - Which does converge!
PageRank

- Using the above, we can also compute the inverse of \((I_n - sM)\) and multiply \(s*e\) by it.

- This gives a way to compute pagerank by a much simpler series of matrix-vector multiplications.

- Recall that if we disallow “teleportation” by setting \(s = 0\), the calculating the pagerank is the power iteration:
  - \(p^k = M*p^{k-1}\)
  - Which is (generally) a sparse product!