MapReduce
Graph Algorithms

Sergei Vassilvitskii
Reminder: MapReduce

Unordered Data

- *(u,v)*: 3
- *(x,v)*: 5
- *(v,w)*: 2
- *(u,x)*: 4
- *(x,w)*: 1
MapReduce (Data View)

Unordered Data

Map

(u,v) 3
(x,v) 5
(v,w) 2
(u,x) 4
(x,w) 1
MapReduce (Data View)

Unordered Data

Map

Shuffle
MapReduce (Data View)

Unordered Data

Map

Shuffle

Reduce
MapReduce (Data View)

Unordered Data

Map

Shuffle

Reduce

Unordered Data
Outline: Graph Algorithms

Dense Graphs
- Connectivity
- Matching

Sparse Graphs
- Pregel/Giraph Model
- Connectivity
- Matchings

Application
- Densest Subgraph
Dense Graphs

Are real world graphs:

- sparse: \( m = \tilde{O}(n) \)
- dense: \( m = n^{1+c} \), for some \( c > 0 \)
Graphs over time

(a) arXiv
(b) Patents
(c) Autonomous Systems
(d) Affiliation network
Find the core of the problem:
- Reduce the problem size in parallel
- Solve the smaller instance sequentially

Roadmap:
- Identify redundant information
- Filter out redundancy to reduce input size
- Solve the smaller problem
Connectivity

Given an undirected graph, find the number of connected components.

Sequential:
- Consider edges one at a time
- Maintain connected components (in a Union Find tree)
Connected Components

Given a graph:
Begin: Each node is a separate component
Connected Components

Begin: Each node is a separate component

With every edge, select one of the colors
Connected Components

Begin: Each node is a separate component
With every edge, select one of the colors
Connected Components

Begin: Each node is a separate component
With every edge, select one of the colors
Update all of the colors in a component
Connected Components

Begin: Each node is a separate component

With every edge, select one of the colors

Update all of the colors in a component
Begin: Each node is a separate component
With every edge, select one of the colors
Update all of the colors in a component
Connected Components

Begin: Each node is a separate component

With every edge, select one of the colors

Update all of the colors in a component

Count the number of colors: 2

[Diagram of connected components]
Connectivity

Given an undirected graph, find the number of connected components.

Sequential:
- Consider edges one at a time
- Maintain connected components (in a Union Find tree)
Connectivity

Given an undirected graph, find the number of connected components.

Sequential:
- Consider edges one at a time
- Maintain connected components (in a Union Find tree)

Filtering:
- What makes an edge redundant?
Connectivity

Given an undirected graph, find the number of connected components.

Sequential:
- Consider edges one at a time
- Maintain connected components (in a Union Find tree)

Filtering:
- What makes an edge redundant?
- If we already know the endpoints are connected
Given a graph:
Given a graph:
1. Partition edges (randomly)
Connected Components

Given a graph:

1. Partition edges (randomly)
2. Summarize (keep $\leq n - 1$ edges per partition)

Machine 1

Machine 2
Connected Components

Given a graph:
1. Partition edges (randomly)
2. Summarize (keep $\leq n - 1$ edges per partition)
3. Recombine
Connected Components

Given a graph:
1. Partition edges (randomly)
2. Summarize (keep \( \leq n - 1 \) edges per partition)
3. Recombine
4. Compute CC’s
Analysis

Given: \( k \) machines:
- Total Runtime: \( T_{cc}(m/k) + T_{cc}(nk) \)
- Memory per machine: \( O(m/k + nk) \)
  - Actually, can stream through edges so \( O(n) \) suffices
- 2 Rounds total
Analysis

Given: $k$ machines:
- Total Runtime: $T_{cc}(m/k) + T_{cc}(nk)$
- Memory per machine: $O(m/k + nk)$
  - Actually, can stream through edges so $O(n)$ suffices
- 2 Rounds total

Notes:
- Semi-streaming model: vertices must fit in memory
- Instead of two passes can achieve a trade-off between memory and number of passes
Matchings

Finding matchings
- Given an undirected graph $G = (V, E)$
- Find a maximum matching
Matchings

Finding matchings
- Given an undirected graph \( G = (V, E) \)
- Find a maximum matching
- Find a maximal matching
Matchings

Finding matchings
- Given an undirected graph \( G = (V, E) \)
- Find a maximum matching
- Find a maximal matching

Try random partitions:
- Find a matching on each partition
- Compute a matching on the matchings
- Does not work: may make very limited progress
Looking for redundancy

Matching:
- Could drop the edge if an endpoint already matched

Idea:
- Find a seed matching (on a sample)
- Remove all ‘dead’ edges
- Recurse on remaining edges
Given a graph:
1. Take a random sample
Algorithm

Given a graph:

1. Take a random sample
Given a graph:
1. Take a random sample
2. Find a maximal matching on sample
Algorithm

Given a graph:
1. Take a random sample
2. Find a maximal matching on sample
3. Look at original graph
Algorithm

Given a graph:
1. Take a random sample
2. Find a maximal matching on sample
3. Look at original graph, drop dead edges
Algorithm

Given a graph:
1. Take a random sample
2. Find a maximal matching on sample
3. Look at original graph, drop dead edges
4. Find matching on remaining edges
Key Lemma:
- Suppose the sampling rate is \( p = \frac{n^{1+c}}{m} \) for some \( c > 0 \).
- Then with high probability the number of edges remaining after the prune step is at most:

\[
\frac{2n}{p} = \frac{2m}{n^c}
\]
Analysis

The sampling rate is: \[ p = \frac{n^{1+c}}{m} \] for some \( c > 0 \)
The sampling rate is: \[ p = \frac{n^{1+c}}{m} \] for some \( c > 0 \)

Suppose some set \( I \subset V \) is unmatched after the prune step
Then \( I \) is an independent set in the sample
Otherwise vertices would have been matched
The sampling rate is: \[ p = \frac{n^{1+c}}{m} \quad \text{for some} \quad c > 0 \]

Suppose some set \( I \subseteq V \) is unmatched after the prune step.
Then \( I \) is an independent set in the sample.
Otherwise vertices would have been matched.

If \(|E[I]| > O(n/p)|\)
i.e. we have a lot of edges left over

Then the probability that none of the edges were picked is at most
\[ (1 - p)^{n/p} \leq e^{-n} \]
The sampling rate is: \( p = \frac{n^{1+c}}{m} \) for some \( c > 0 \)

Suppose some set \( I \subset V \) is unmatched after the prune step
Then \( I \) is an independent set in the sample
Otherwise vertices would have been matched

If \( |E[I]| > O(n/p) \)
i.e. we have a lot of edges left over

Then the probability that none of the edges were picked is at most
\[
(1 - p)^{n/p} \leq e^{-n}
\]

The total possible number of such sets \( I \) is \( 2^n \)

Thus the total probability of a bad event (too many edges left over) is:
\[
2^n \cdot e^{-n} \leq 0.75^n
\]
Analysis

Key Lemma:
- Suppose the sampling rate is $p = \frac{n^{1+c}}{m}$ for some $c > 0$.
- Then with high probability the number of edges remaining after the prune step is at most:

$$\frac{2n}{p} = \frac{2m}{n^c}$$

Corollaries:
- Given $n^{1+c}$ memory, algorithm requires $O(1)$ rounds
- Given $O(n \log n)$ memory, algorithm requires $O\left(\frac{\log n}{\log \log n}\right)$ rounds.
- PRAM simulations: $\Theta(\log n)$ rounds
Outline: Graph Algorithms

Dense Graphs
- Connectivity
- Matching

Sparse Graphs
- Pregel Model
- Connectivity
- Matchings

Application
- Densest Subgraph
Optimizing Graphs

Computation:
- Most often computation is along the edges
- Djikstra’s shortest path algorithm

Data:
- Graph itself usually does not change
- Pass values around vertices
Optimizations:
- Partition the graph once across the machines
- Keep the graph structure local (don’t shuffle it!)
Optimizations:
- Partition the graph once across the machines
- Keep the graph structure local (don’t shuffle it!)

Implications:
- Vertex central view of the data
- Each round:
  - Each vertex collects all messages sent to it
  - Send messages to its neighbors
  - Can also modify the graph
Optimizations:
- Partition the graph once across the machines
- Keep the graph structure local (don’t shuffle it!)

Implications:
- Vertex central view of the data
- Each round:
  - Each vertex collects all messages sent to it
  - Send messages to its neighbors
  - Can also modify the graph
- Under the covers:
  - Vertices act as a key
  - Edges are stored locally with each vertex, reducing shuffle time
for each vertex v:
    for every message received: m
        if (value > m)
            value = m;
        if (value changed)
            SendMessageToAllNeighbors(value + 1);
Example: BFS

for each vertex v:
    for every message received: m
        if (value > m)
            value = m;
        if (value changed)
            SendMessageToAllNeighbors(value + 1);
Example: BFS

for each vertex v:
    for every message received: m
        if (value > m)
            value = m;
        if (value changed)
            SendMessageToAllNeighbors(value + 1);
Example: BFS

for each vertex \( v \):
  for every message received: \( m \)
    if (value > m)
      value = m;
    if (value changed)
      SendMessageToAllNeighbors(value + 1);
Example: BFS

for each vertex v:
  for every message received: m
    if (value > m)
      value = m;
    if (value changed)
      SendMessageToAllNeighbors(value + 1);
Example: BFS

for each vertex $v$:
  for every message received: $m$
    if (value > $m$)
      value = $m$;
    if (value changed)
      SendMessageToAllNeighbors(value + 1);
Example: BFS

for each vertex v:
    for every message received: m
        if (value > m)
            value = m;
    if (value changed)
        SendMessageToAllNeighbors(value + 1);
Example: BFS

for each vertex $v$:
   for every message received: $m$
      if (value > $m$)
         value = $m$;
      if (value changed)
         SendMessageToAllNeighbors(value + 1);
Example: BFS

for each vertex v:
  for every message received: m
    if (value > m)
      value = m;
    if (value changed)
      SendMessageToAllNeighbors(value + 1);
Beyond BFS

Simple Algorithms:
- BFS,
- shortest paths (single source & all pairs)
- ...

What about:
- connectivity?
- matchings?
- ...
Connectivity, Try 1:

- Begin with a unique id at every node
- In each super-round:
  - Every node identifies the minimum in its 2-neighborhood
  - Adds edges from all neighbors at least as big to the minimum
  - Sets own id to the minimum
Connectivity, Try 1:
- Begin with a unique id at every node
- In each super-round:
  - Every node identifies the minimum in its 2-neighborhood
  - Adds edges from all neighbors at least as big to the minimum
  - Sets own id to the minimum

![Diagram of connectivity numbers: 5, 4, 3, 2, 6, 1]
Connectivity, Try 1:
- Begin with a unique id at every node
- In each super-round:
  - Every node identifies the minimum in its 2-neighborhood
  - Adds edges from all neighbors at least as big to the minimum
  - Sets own id to the minimum
Connectivity, Try 1:
- Begin with a unique id at every node
- In each super-round:
  - Every node identifies the minimum in its 2-neighborhood
  - Adds edges from all neighbors at least as big to the minimum
  - Sets own id to the minimum
Connectivity, Try 1:
- Begin with a unique id at every node
- In each super-round:
  - Every node identifies the minimum in its 2-neighborhood
  - Adds edges from all neighbors at least as big to the minimum
  - Sets own id to the minimum
Connectivity, Try 1:
- Begin with a unique id at every node
- In each super-round:
  - Every node identifies the minimum in its 2-neighborhood
  - Adds edges from all neighbors at least as big to the minimum
  - Sets own id to the minimum

![Diagram showing the process of connectivity](image)
Connectivity, Try 1:
- Begin with a unique id at every node
- In each super-round:
  - Every node identifies the minimum in its 2-neighborhood
  - Adds edges from all neighbors at least as big to the minimum
  - Sets own id to the minimum

![Diagram illustrating the connectivity algorithm](image-url)
Connectivity

Connectivity, Try 1:
- Begin with a unique id at every node
- In each super-round:
  - Every node identifies the minimum in its 2-neighborhood
  - Adds edges from all neighbors at least as big to the minimum
  - Sets own id to the minimum
- Analysis:
  - Takes $O(\log n)$ rounds to complete
  - Add $O(n)$ edges per round
Connectivity, Try 2:

- Begin with a unique id at every node
- In each super-round:
  - Every node identifies the minimum in its 2-neighborhood
  - Adds edge from itself to the minimum
  - Sets own id to the minimum

- Conjecture
  - This takes also $O(\log n)$ rounds to complete
Sparse Matchings

An example of adapting the spirit of a PRAM model

- Leads to an $O(\log n)$ algorithm
- With very simple computations per round
- Can be implemented either in MapReduce or in the Congest model
- Due to Israeli & Itai, 1986
Each Super-Round:
- Each Node picks one neighboring edge, directed away
Each Super-Round:
- Each Node picks one neighboring edge, directed away
Algorithm

Each Super-Round:
- Each Node picks one neighboring edge, directed away
- Nodes with in-degree > 1, pick one edge at random
Each Super-Round:
- Each Node picks one neighboring edge, directed away
- Nodes with in-degree > 1, pick one edge at random
Algorithm

Each Super-Round:
- Each Node picks one neighboring edge, directed away
- Nodes with in-degree > 1, pick one edge at random
- Nodes of degree 2 select one edge at random, those of degree 1 select their neighboring edge
Each Super-Round:
- Each Node picks one neighboring edge, directed away
- Nodes with in-degree > 1, pick one edge at random
- Nodes of degree 2 select one edge at random, those of degree 1 select their neighboring edge
Algorithm

Each Super-Round:
- Each Node picks one neighboring edge, directed away
- Nodes with in-degree > 1, pick one edge at random
- Nodes of degree 2 select one edge at random, those of degree 1 select their neighboring edge
- If both endpoints selected same edge, add it to the matching
Algorithm

Each Super-Round:
- Each Node picks one neighboring edge, directed away
- Nodes with in-degree > 1, pick one edge at random
- Nodes of degree 2 select one edge at random, those of degree 1 select their neighboring edge
- If both endpoints selected same edge, add it to the matching
Algorithm

Each Super-Round:
- Each Node picks one neighboring edge, directed away
- Nodes with in-degree > 1, pick one edge at random
- Nodes of degree 2 select one edge at random, those of degree 1 select their neighboring edge
- If both endpoints selected same edge, add it to the matching

Recurse on unmatched nodes
Each Super-Round:
- Each Node picks one neighboring edge, directed away
- Nodes with in-degree > 1, pick one edge at random
- Nodes of degree 2 select one edge at random, those of degree 1 select their neighboring edge
- If both endpoints selected same edge, add it to the matching

Recurse on unmatched nodes
Sparse Graphs Conclusion

When nodes don’t fit into memory
- Very different from Streaming algorithms
- Possible to adapt PRAM algorithms
- Many open questions!
Applications

Back to Social Graph Mining

- Yesterday: Finding tight knit communities
- Today: Finding large communities
Problem: Given a graph \( G = (V, E) \), find \( V' \subseteq V \) that maximizes:

\[
\rho = \frac{|E(V')|}{|V'|}
\]
Finding Densest Subgraph

Problem: Given a graph $G = (V, E)$, find $V' \subseteq V$ that maximizes:

$$\rho = \frac{|E(V')|}{|V'|}$$

Useful Primitive in Graph Analysis:
- Community Detection
- Graph Compression
- Link SPAM Mining
- Many other applications
Finding Densest Subgraph

Problem: Given a graph \( G = (V, E) \), find \( V' \subseteq V \) that maximizes:

\[
\rho = \frac{|E(V')|}{|V'|}
\]

Useful Primitive in Graph Analysis

Can be solved exactly:
- LP Formulation
- Multiple Max flow computations
Finding Densest Subgraphs

Simple Algorithm [Charikar ’00]:
- Iteratively remove the lowest degree node and update vertex degrees
- Keep the densest intermediate subgraph
Finding Dense Subgraphs

Simple Algorithm:
- Iteratively remove the lowest degree node and update vertex degrees
- Keep the densest intermediate subgraph

Best Density: 16/11
Current Density: 16/11
Finding Dense Subgraphs

Simple Algorithm:
- Iteratively remove the lowest degree node and update vertex degrees
- Keep the densest intermediate subgraph

Best Density: 16/11
Current Density: 16/11
Finding Dense Subgraphs

Simple Algorithm:
- Iteratively remove the lowest degree node and update vertex degrees
- Keep the densest intermediate subgraph

Best Density: 15/10
Current Density: 15/10
Finding Dense Subgraphs

Simple Algorithm:
- Iteratively remove the lowest degree node and update vertex degrees
- Keep the densest intermediate subgraph

Best Density: 14/9
Current Density: 14/9
Simple Algorithm:
- Iteratively remove the lowest degree node and update vertex degrees
- Keep the densest intermediate subgraph
Finding Dense Subgraphs

Simple Algorithm:
- Iteratively remove the lowest degree node and update vertex degrees
- Keep the densest intermediate subgraph

Best Density: $\frac{12}{7}$
Current Density: $\frac{12}{7}$
Finding Dense Subgraphs

Best Density: 12/7
Current Density: 10/6

Simple Algorithm:
- Iteratively remove the lowest degree node and update vertex degrees
- Keep the densest intermediate subgraph
Finding Dense Subgraphs

Simple Algorithm:
- Iteratively remove the lowest degree node and update vertex degrees
- Keep the densest intermediate subgraph

Best Density: 9/5
Current Density: 9/5
Finding Dense Subgraphs

Simple Algorithm:
- Iteratively remove the lowest degree node and update vertex degrees
- Keep the densest intermediate subgraph

Best Density: 9/5
Current Density: 6/4
Finding Dense Subgraphs

Simple Algorithm:
- Iteratively remove the lowest degree node and update vertex degrees
- Keep the densest intermediate subgraph

Best Density: 9/5
Current Density: 3/3
Finding Dense Subgraphs (Analysis)

Approximation Ratio:
- Guaranteed to return a 2-approximation

Proof:
- Let $V^* \subseteq V$ be the optimal solution, and $\lambda^* = \frac{|E[V^*]|}{|V^*|}$ the optimal density.
- Consider the first time a vertex from $V^*$ is removed.
- Every vertex in $V^*$ has degree at least $\lambda^*$.
  - Otherwise can improve optimum density
- Therefore the density of that subgraph is at least:
  \[
  \frac{\lambda^*|V^*|}{2|V^*|} = \frac{\lambda^*}{2}
  \]
Finding Dense Subgraphs (Analysis)

Approximation Ratio:
- Guaranteed to return a 2-approximation

Running Time:
- RAM:
  - Maintain a heap on vertex degrees
  - Update keys upon removing every edge
  - Straightforward implementation in $O(m \log n)$
- Streaming:
  - Seemingly need one pass per vertex to adapt this algorithm
  - Can show that need $\Omega(n / \log n)$ memory if using $O(\log n)$ passes
- MapReduce?
  - Open question in Chierichetti, Kumar and Tompkins WWW ’10.
Parallel Dense Subgraphs

Sequential Algorithm:
- Remove the node with the smallest degree
Parallel Dense Subgraphs

Sequential Algorithm:
- Remove the node with the smallest degree

Parallel Version:
- Remove all nodes with less degree less than $(1 + \epsilon) \times$ average degree
- Of course this also includes the smallest degree node

- Every Step:
  • Round 1: Count remaining edges, vertices, compute vertex degrees
    - Distributed counting
  • Round 2: Remove vertices with degree below threshold
    - Distributed checking
Parallel Dense Subgraphs

Parallel Algorithm:
- Iteratively remove nodes with degree below average and update vertex degrees
- Keep the densest intermediate subgraph

Best Density: 16/11
Current Density: 16/11
Average Degree: 32/11
Parallel Dense Subgraphs

Parallel Algorithm:
- Iteratively remove nodes with degree below average and update vertex degrees
- Keep the densest intermediate subgraph

Best Density: 16/11
Current Density: 16/11
Average Degree: 32/11
Parallel Dense Subgraphs

Parallel Algorithm:
- Iteratively remove nodes with degree below average and update vertex degrees
- Keep the densest intermediate subgraph

Best Density: 9/5
Current Density: 9/5
Average Degree: 18/5
Parallel Dense Subgraphs

Parallel Algorithm:
- Iteratively remove nodes with degree below average and update vertex degrees
- Keep the densest intermediate subgraph

Best Density: 9/5
Current Density: 3/3
Average Degree: 6/3
Algorithm:
- Each round remove all vertices with degree less than \( (1 + \epsilon) \times \text{average} \).

How many vertices do we remove?
- One cannot have too many vertices above average (This is not Lake Wobegon)
- Easy [Markov inequality]: at most a \( \frac{1}{1 + \epsilon} \) fraction of vertices remains in every round.

- Therefore algorithm terminates after \( O \left( \frac{1}{\epsilon \log n} \right) \) rounds.
Parallel Densest Subgraph (Analysis)

Algorithm:
- Each round remove all vertices with degree less than \((1 + \epsilon) \times \text{average}\).

How many vertices do we remove?
- One cannot have too many vertices above average (This is not Lake Wobegon)
- Easy [Markov inequality]: at most a \(\frac{1}{1 + \epsilon}\) fraction of vertices remains in every round.

- Therefore algorithm terminates after \(O\left(\frac{1}{\epsilon \log n}\right)\) rounds

Approximation Ratio:
- Achieves a \((2 + \epsilon)\) approximation in the worst case
  - Only look at the degree of the nodes removed as compared to average.
How well does it work?

- Quickly reduce the size of the graph.
- Approximation ratio between 1.06 and 1.4 at $\varepsilon = 1$
Overall

Improving the sequential algorithm:

- Original algorithm: $O(m)$ heap updates:
  - Update vertex degrees every time an edge is removed.

- New algorithm $O(n)$ heap updates:
  - Number of vertices decreases geometrically every round
Wrap Up

Graphs:
- At the core of many large data computations
- Many follow heavy tailed degree distributions
- Dense: Sample & Prune leads to fast algorithms
- Sparse: Adapt PRAM Algorithms
Wrap Up

Graphs:
- At the core of many large data computations
- Many follow heavy tailed degree distributions
- Dense: Sample & Prune leads to fast algorithms
- Sparse: Adapt PRAM Algorithms

Next Up:
- Clustering & Machine Learning
References


- Pregel: A System for Large-Scale Graph Processing. Grzegorz Malewicz, Matthew Austern, Aart Bik, James Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski, SIGMOD 2010.


- Densest Subgraph in Streaming and MapReduce. Bahman Bahmani, Ravi Kumar, S.V., VLDB 2012.