Graphs

**Graph** $G = (V,E)$
- a set of **vertices** and a set of **edges** between vertices

$n = |V|$ (number of vertices)

$m = |E|$ (number of edges)

$D =$ diameter (max #hops between any pair of vertices)

- Edges can be directed or undirected, weighted or not.
- They can even have attributes (i.e. semantic graphs)
- Sequences of edges $<u_1, u_2>, <u_2, u_3>, \ldots, <u_{n-1}, u_n>$ is a **path** from $u_1$ to $u_n$. Its **length** is the sum of its weights.
Routing in transportation networks

Road networks, Point-to-point shortest paths: 15 seconds (naïve) → 10 microseconds

Internet and the WWW

• The world-wide web can be represented as a directed graph
  – Web search and crawl: traversal
  – Link analysis, ranking: Page rank and HITS
  – Document classification and clustering

• Internet topologies (router networks) are naturally modeled as graphs
Large-scale data analysis

• Graph abstractions are very useful to analyze complex data sets.
• Sources of data: simulations, experimental devices, the Internet, sensor networks
• Challenges: data size, heterogeneity, uncertainty, data quality

Astrophysics: massive datasets, temporal variations

Bioinformatics: data quality, heterogeneity

Social Informatics: new analytics challenges, data uncertainty

Image sources: (1) http://physics.nmt.edu/images/astro/hst_starfield.jpg (2,3) www.visualComplexity.com
Large Graphs in Biology

Whole genome assembly

A  Read Layout

B  Overlap Graph

Vertices: reads

C  de Bruijn Graph

Vertices: k-mers

26 billion (8B of which are non-erroneous) unique k-mers (vertices) in the hexaploid wheat genome W7984 for k=51

Schatz et al. (2010) Perspective: Assembly of Large Genomes w/2nd-Gen Seq. Genome Res. (figure reference)
Large Graphs in Biology

Whole genome assembly

A Read Layout

R_1: GACCTACA
R_2: ACCTACAA
R_3: CCTACAAG
R_4: CTACAAGT
A: TACAGGT
B: ACAAGTTA
C: CAAGTTAG
X: TACAGTC
Y: ACAAGTCG
Z: CAAGTCCG

B Overlap Graph

Vertices: reads

C de Bruijn Graph

Vertices: k-mers

26 billion (8B of which are non-erroneous) unique k-mers (vertices) in the hexaploit wheat genome W7984 for k=51

Graph Theoretical analysis of Brain Connectivity

Potentially millions of neurons and billions of edges with developing technologies

Schatz et al. (2010) Perspective: Assembly of Large Genomes w/2nd-Gen Seq. Genome Res. (figure reference)
Adjacency List graph representation
Graph Algorithms

• Traversals
  • DFS, BFS

• Finding paths
  • Single-source shortest paths (Dijkstra, Bellman-Ford)
  • All-pairs shortest-paths (Floyd-Warshall)

• Maximal independent sets

• Decomposition (connected components, strongly connected components)

• Maximum cardinality matching

• Connecting
  • Minimum spanning tree
Spanning Tree Definition

- A spanning tree, $T$, of a connected undirected graph $G$ is:
  - rooted at some vertex of $G$
  - defined by a parent map for each vertex
  - contains all the vertices of $G$, i.e. spans all vertices
  - contains exactly $|v| - 1$ edges
    - adding any other edge will create a cycle
  - contains no cycles (a tree!)
- The edges involved in $T$ are a subset of the edges in $G$
An Example Graph with 4 possible spanning trees rooted at vertex A

Example Undirected Graph:

Spanning Trees (edges are directed from child to parent):

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>null</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>null</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>null</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>null</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
</tr>
</tbody>
</table>
Sequential Spanning Tree Algorithm

1. class V {
2.     V [] neighbors; // adjacency list for input graph
3.     V parent; // output value of parent in spanning tree
4.     boolean makeParent(V n) {
5.         if (parent == null) { parent = n; return true; }
6.         else return false; // return true if n became parent
7.     } // makeParent
8.     void compute() {
9.         for (int i=0; i<neighbors.length; i++) {
10.            final V child = neighbors[i];
11.            if (child.makeParent(this))
12.                child.compute(); // recursive call
13.        }
14.     } // compute
15. } // class V
16. . . . // main program
17. root.parent = root; // Use self-cycle to identify root
18. root.compute();
19. . . .
Exercise: Parallel Spanning Tree Algorithm using object-based isolated construct

1. class V {
2.   V [] neighbors; // adjacency list for input graph
3.   V parent; // output value of parent in spanning tree
4.   boolean makeParent(V n) {
5.     if (parent == null) { parent = n; return true; }
6.     else return false; // return true if n became parent
7.   } // makeParent
8.   void compute() {
9.     for (int i=0; i<neighbors.length; i++) {
10.        final V child = neighbors[i];
11.        if (child.makeParent(this))
12.           child.compute(); // recursive call
13.     }
14. } // compute
15. } // class V
16. . . . // main program
17. root.parent = root; // Use self-cycle to identify root
18. root.compute();
19. . . .
Exercise: Parallel Spanning Tree Algorithm using object-based isolated construct

1. class V {
2.     V [] neighbors; // adjacency list for input graph
3.     V parent; // output value of parent in spanning tree
4.     boolean makeParent(final V n) {
5.         return isolatedWithReturn(this, () -> {
6.             if (parent == null) { parent = n; return true; }
7.             else return false; // return true if n became parent
8.         });
9.     } // makeParent
10.    void compute() {
11.        for (int i=0; i<neighbors.length; i++) {
12.            final V child = neighbors[i];
13.            if (child.makeParent(this))
14.                async(() -> { child.compute(); });
15.        }
16.    } // compute
17. } // class V
18. ...
19. root.parent = root; // Use self-cycle to identify root
20. finish(() -> { root.compute(); });
21. ...
Minimum Spanning Tree

- For graphs that have edge weights
- Spanning tree with a minimum weight
- Sequential algorithms:
  - Prim’s algorithm: greedy, grow a single tree by adding nodes closest to it
  - Kruskal’s algorithm: greedy, add lightest edges that don’t create a cycle
  - Boruvka’s algorithm: combination of Prim’s and Kruskal’s
    - Can be parallelized
Starting from empty T, choose a vertex at random and initialize

\[ V = \{1\}, \ E' = \{\} \]
Prim’s Algorithm

Choose the vertex \( u \) not in \( V \) such that edge weight from \( u \) to a vertex in \( V \) is minimal (greedy!)

\( V = \{1, 3\} \) \( E' = \{(1, 3)\} \)
Prim’s Algorithm

Repeat until all vertices have been chosen

Choose the vertex \( u \) not in \( V \) such that edge weight from \( v \) to a vertex in \( V \) is minimal (greedy!)

\[ V = \{1,3,4\} \quad E' = \{(1,3),(3,4)\} \]

\[ V = \{1,3,4,5\} \quad E' = \{(1,3),(3,4),(4,5)\} \]

....

\[ V = \{1,3,4,5,2,6\} \]

\[ E' = \{(1,3),(3,4),(4,5),(5,2),(2,6)\} \]
Prim’s Algorithm

Repeat until all vertices have been chosen

\[ V = \{1,3,4,5,2,6\} \]

\[ E' = \{(1,3),(3,4),(4,5),(5,2),(2,6)\} \]

Final Cost: \(1 + 3 + 4 + 1 + 1 = 10\)
Kruskal’s Algorithm

- Select edges in order of increasing cost
- Accept an edge to expand tree or forest only if it does not cause a cycle
- Implementation using adjacency list, priority queues and disjoint sets
Kruskal’s Algorithm
Kruskal’s Algorithm
Kruskal’s Algorithm

Diagram of a graph with weighted edges.
Kruskal’s Algorithm
Kruskal’s Algorithm
Kruskal’s Algorithm
Boruvka’s Algorithm

• Combination of Prim’s and Kruskal’s
• Grow a tree (component) by picking the lightest edge connected to it, just like Prim
• Connect the trees when the lightest edge is between them, just like Kruskal
• Growing of each tree can be done in parallel
• Component contraction
  • Each component represented by a single node
  • When connecting two components, contract the edge and make a single node to represent the two
Boruvka’s Algorithm

Animation: Randy Cornell, Texas State University
Boruvka’s Algorithm

Animation: Randy Cornell, Texas State University
Parallel Boruvka’s Algorithm

- Java threads or async tasks picking up components off the worklist
  - You don’t want too many threads of tasks, tune for the machine
  - Worklist has to allow concurrent access
- Grow components in parallel
- When inspecting the closest node to expand the component, have to synchronize
  - Other thread or task could be also accessing it
  - Careful not to introduce deadlock
- When contracting an edge, have to synchronize
- When there’s only a single component left, you are done