Outline

• What is a graph?
• Graph representations
• Implementing breadth-first search
• Graph compression/reordering
What is a graph?

- Vertices model objects
- Edges model relationships between objects

Image courtesy of STRING. Used under CC-BY.
What is a graph?

• **Edges can be directed**
  • Relationship can go one way or both ways

Image created by MIT OpenCourseWare.
What is a graph?

- Edges can be weighted
  - Denotes “strength”, distance, etc.

Distance between cities

Flight costs

© 2008–2018 by the MIT 6.172 Lecturers
What is a graph?

• Vertices and edges can have types and metadata

Google Knowledge Graph
SOME MORE APPLICATIONS OF GRAPHS
Social network queries

• **Examples:**
  - Finding all your friends who went to the same high school as you
  - Finding common friends with someone
  - Social networks recommending people whom you might know
  - Product recommendation
Finding good clusters

- Some applications
  - Finding people with similar interests
  - Detecting fraudulent websites
  - Document clustering
  - Unsupervised learning

- Finding groups of vertices that are “well-connected” internally and “poorly-connected” externally
More Applications

Connectomics
Image courtesy of Andreas Horn. Used under CC-BY.

- Study of the brain network structure

Image Segmentation

- Pixels correspond to vertices
- Edges between neighboring pixels with weight corresponding to similarity
GRAPH REPRESENTATIONS
Graph Representations

- Vertices labeled from 0 to n-1

```
0 1 0 0 0
1 0 0 1 1
0 0 0 1 0
0 1 1 0 0
0 1 0 0 0
```

Adjacency matrix

(“1” if edge exists, “0” otherwise)

- What is the space requirement for each in terms of number of edges (m) and number of vertices (n)?
Graph Representations

- **Adjacency list**
  - Array of pointers (one per vertex)
  - Each vertex has an unordered list of its edges

- **What is the space requirement?**
- **Can substitute linked lists with arrays for better cache performance**
  - Tradeoff: more expensive to update graph
Graph Representations

- Compressed sparse row (CSR)
  - Two arrays: Offsets and Edges
  - Offsets[i] stores the offset of where vertex i’s edges start in Edges

Vertex IDs  0  1  2  3
Offsets  0  4  5  11  ...
Edges  2  7  9  16  0  1  6  9  12  ...

- How do we know the degree of a vertex?
- Space usage?
- Can also store values on the edges with an additional array or interleaved with Edges
### Tradeoffs in Graph Representations

**What is the cost of different operations?**

<table>
<thead>
<tr>
<th></th>
<th>Adjacency matrix</th>
<th>Edge list</th>
<th>Adjacency list</th>
<th>Compressed sparse row</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Storage cost / scanning whole graph</strong></td>
<td>O(n²)</td>
<td>O(m)</td>
<td>O(m+n)</td>
<td>O(m+n)</td>
</tr>
<tr>
<td><strong>Add edge</strong></td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)/O(deg(v))</td>
<td>O(m+n)</td>
</tr>
<tr>
<td><strong>Delete edge from vertex v</strong></td>
<td>O(1)</td>
<td>O(m)</td>
<td>O(deg(v))</td>
<td>O(m+n)</td>
</tr>
<tr>
<td><strong>Finding all neighbors of a vertex v</strong></td>
<td>O(n)</td>
<td>O(m)</td>
<td>O(deg(v))</td>
<td>O(deg(v))</td>
</tr>
<tr>
<td><strong>Finding if w is a neighbor of v</strong></td>
<td>O(1)</td>
<td>O(m)</td>
<td>O(deg(v))</td>
<td>O(deg(v))</td>
</tr>
</tbody>
</table>

- There are variants/combinations of these representations
Graph Representations

• The algorithms we will discuss today are best implemented with compressed sparse row (CSR) format
  • Sparse graphs
  • Static algorithms—no updates to graph
  • Need to scan over neighbors of a given set of vertices
Properties of real–world graphs

- They can be big (but not too big)
- Sparse (m much less than n^2)
- Degrees can be highly skewed

Studies have shown that many real–world graphs have a power law degree distribution

\[ \#\text{vertices with deg. } d \approx a \times d^{-p} \]
\(2 < p < 3\)
IMPLEMENTING A GRAPH ALGORITHM: BREADTH-FIRST SEARCH
Breadth-First Search (BFS)

- Given a source vertex $s$, visit the vertices in order of distance from $s$.

Possible outputs:

- Vertices in the order they were visited
  - D, B, C, E, A

- The distance from each vertex to $s$
  
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- A BFS tree, where each vertex has a parent to a neighbor in the previous level

**Applications**

- Betweenness centrality
- Eccentricity estimation
- Maximum flow
- Web crawlers
- Network broadcasting
- Cycle detection
  ...
Serial BFS Algorithm

Breadth-First-Search(Graph, root):

   for each node n in Graph:
       n.distance = INFINITY
       n.parent = NIL

Source: https://en.wikipedia.org/wiki/Breadth-first_search
Serial BFS Algorithm

- Assume graph is given in compressed sparse row format
- Two arrays: Offsets and Edges
- \( n \) vertices and \( m \) edges (assume \( \text{Offsets}[n] = m \))

```c
int* parent = (int*) malloc(sizeof(int)*n);
int* queue = (int*) malloc(sizeof(int)*n);

for(int i=0; i<n; i++) {
    parent[i] = -1;
}

queue[0] = source;
parent[source] = source;

int q_front = 0, q_back = 1;

//while queue not empty
while(q_front != q_back) {
    int current = queue[q_front++]; //dequeue
    int degree = Offsets[current+1]-Offsets[current];
    for(int i=0; i<degree; i++) {
        int ngh = Edges[Offsets[current]+i];
        //check if neighbor has been visited
        if(parent[ngh] == -1) {
            queue[0] = source; parent[ngh] = current;
            parent[source] = source;
            //enqueue neighbor
            queue[q_back++] = ngh;
        }
    }
}
```

- What is the most expensive part of the code?
  - Random accesses cost more than sequential accesses
Analyzing the program

```c
int* parent = (int*) malloc(sizeof(int)*n);
int* queue = (int*) malloc(sizeof(int)*n);

for(int i=0; i<n; i++) {
    parent[i] = -1;
}

queue[0] = source;
parent[source] = source;

int q_front = 0; q_back = 1;

while(q_front != q_back) {
    int current = queue[q_front++]; // dequeue
    int degree = Offsets[current+1]-Offsets[current];
    for(int i=0;i<degree; i++) {
        int ngh = Edges[Offsets[current]+i];
        //check if neighbor has been visited
        if(parent[ngh] == -1) {
            parent[ngh] = current;
            //enqueue neighbor
            queue[q_back++] = ngh;
        }
    }
}
```

- (Approx.) analyze number of cache misses (cold cache; cache size << n; 64 byte cache line size; 4 byte int)
  - n/16 for initialization
  - n/16 for dequeueing
  - n for accessing Offsets array
  - ≤ 2n + m/16 for accessing Edges array
  - m for accessing parent array

Total ≤ (51/16)n + (17/16)m
Analyzing the program

```c
int* parent = (int*) malloc(sizeof(int)*n);
int* queue = (int*) malloc(sizeof(int)*n);

for(int i=0; i<n; i++) {
    parent[i] = -1;
}

queue[0] = source;
parent[source] = source;

int q_front = 0; q_back = 1;
```

```c
//while queue not empty
while(q_front != q_back) {
    int current = queue[q_front++]; //dequeue
    int degree = Offsets[current+1]-Offsets[current];
    for(int i=0;i<degree; i++) {
        int ngh = Edges[Offsets[current]+i];
        //check if neighbor has been visited
        if(parent[ngh] == -1) {
            queue[q_back++] = ngh;
            //enqueue neighbor
            parent[ngh] = current;
        }
    }
}
```

- What if we can fit a bitvector of size n in cache?
  - Might reduce the number of cache misses
  - More computation to do bit manipulation

Check bitvector first before accessing parent array

\textit{n cache misses instead of m}
BFS with bitvector

```c
int* parent =
    (int*) malloc(sizeof(int)*n);
int* queue =
    (int*) malloc(sizeof(int)*n);
int nv = 1+n/32;
int* visited =
    (int*) malloc(sizeof(int)*nv);

for(int i=0; i<n; i++) {
    parent[i] = -1;
}

for(int i=0; i<nv; i++) {
    visited[i] = 0;
}

queue[0] = source;
parent[source] = source;
visited[source/32] = (1 << (source % 32));

int q_front = 0; q_back = 1;
```

```c
while(q_front != q_back) {
    int current = queue[q_front++]; //dequeue
    int degree =
        Offsets[current+1]-Offsets[current];
    for(int i=0;i<degree; i++) {
        int ngh = Edges[Offsets[current]+i];
        //check if neighbor has been visited
        if(!((1 << ngh%32) & visited[ngh/32])){
            visited[ngh/32] |= (1 << (ngh%32));
            parent[ngh] = current;
            //enqueue neighbor
            queue[q_back++] = ngh;
        }
    }
}
```

- Bitvector version is faster for large enough values of m
PARALLELIZING BREADTH-FIRST SEARCH
Parallel BFS Algorithm

- Can process each frontier in parallel
  - Parallelize over both the vertices and their outgoing edges
- Races, load balancing
Parallel BFS Code

BFS(Offsets, Edges, source) {
  parent, frontier, frontierNext, and degrees are arrays
  cilk_for(int i=0; i<n; i++) parent[i] = -1;
  frontier[0] = source, frontierSize = 1, parent[source] = source;

  while(frontierSize > 0) {
    cilk_for(int i=0; i<frontierSize; i++)
      degrees[i] = Offsets[frontier[i]+1] – Offsets[frontier[i]];
    perform prefix sum on degrees array
    cilk_for(int i=0; i<frontierSize; i++) {
      v = frontier[i], index = degrees[i], d = Offsets[v+1]–Offsets[v];
      for(int j=0; j<d; j++) { //can be parallel
        ngh = Edges[Offsets[v]+j];
        if(parent[ngh] == -1 && compare–and–swap(&parent[ngh], -1, v)) {
          frontierNext[index+j] = ngh;
        } else { frontierNext[index+j] = -1; }
      }
    }
  }

  filter out “-1” from frontierNext, store in frontier, and update frontierSize to be
  the size of frontier (all done using prefix sum)
}

© 2008–2018 by the MIT 6.172 Lecturers
BFS Work–Span Analysis

- Number of iterations \( \leq \) diameter \( D \) of graph
- Each iteration takes \( \Theta(\log m) \) span for cilk_for loops, prefix sum, and filter (assuming inner loop is parallelized)

\[
\text{Span} = \Theta(D \log m)
\]

- Sum of frontier sizes = \( n \)
- Each edge traversed once \( \rightarrow \) \( m \) total visits
- Work of prefix sum on each iteration is proportional to frontier size \( \rightarrow \) \( \Theta(n) \) total
- Work of filter on each iteration is proportional to number of edges traversed \( \rightarrow \) \( \Theta(m) \) total

\[
\text{Work} = \Theta(n+m)
\]
Performance of Parallel BFS

- Random graph with $n=10^7$ and $m=10^8$
  - 10 edges per vertex
- 40-core machine with 2-way hyperthreading

- 31.8x speedup on 40 cores with hyperthreading
- Serial BFS is 54% faster than parallel BFS on 1 thread
Golden Rule of Parallel Programming

Never write nondeterministic parallel programs.

They can exhibit anomalous behaviors, and it’s hard to debug them.
Silver Rule of Parallel Programming

*Never* write nondeterministic parallel programs — *but if you must* — always devise a test strategy to control the nondeterminism!

Typical test strategies
- Turn off nondeterminism.
- Encapsulate nondeterminism.
- Substitute a deterministic alternative.
- Use analysis tools.

*E.g., for performance reasons.*
Dealing with nondeterminism

BFS(Offsets, Edges, source) {
    parent, frontier, frontierNext, and degrees are arrays
    cilk_for(int i=0; i<n; i++) parent[i] = -1;
    frontier[0] = source, frontierSize = 1, parent[source] = source;

    while(frontierSize > 0) {
        cilk_for(int i=0; i<frontierSize; i++)
            degrees[i] = Offsets[frontier[i]+1] – Offsets[frontier[i]];
        perform prefix sum on degrees array
        cilk_for(int i=0; i<frontierSize; i++) {
            v = frontier[i], index = degrees[i], d = Offsets[v+1]–Offsets[v];
            for(int j=0; j<d; j++) {
                ngh = Edges[Offsets[v]+j];
                if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
                    frontierNext[index+j] = ngh;
                } else { frontierNext[index+j] = -1; }
            }
        }
        filter out “–1” from frontierNext, store in frontier, and update frontierSize to be the size of frontier (all done using prefix sum)
    }
}
Deterministic parallel BFS

writeMin(addr, newval):
  oldval = *addr
  while(newval < oldval):
    if(CAS(addr, oldval, newval)): return
    else: oldval = addr*

cilk_for(int i=0; i<frontierSize; i++) {
  v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
  for(int j=0; j<d; j++) {
    ngh = Edges[Offsets[v]+j];
    writeMin(&parent[ngh], v);
  }
}
cilk_for(int i=0; i<frontierSize; i++) {
  v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
  for(int j=0; j<d; j++) {
    ngh = Edges[Offsets[v]+j];
    if(parent[ngh] == v) {
      parent[ngh] = -v; //to avoid revisiting
      frontierNext[index+j] = ngh;
    } else {
      frontierNext[index+j] = -1;
    }
  }
  filter out "-1" from frontierNext, store in frontier, and update frontierSize

On 32 cores, (an optimized version of) deterministic BFS is 5—20% slower than nondeterministic BFS

Smallest value gets written

Check if "won"
DIRECTION–OPTIMIZING
BREADTH–FIRST SEARCH
Growth of frontiers

- For many graphs, frontier grows rapidly and then shrinks
- Most of the work done with frontier (and sum of out-degrees) is large
Two ways to do BFS

- Bottom-up is better when frontier is large and many vertices have been visited
  - Reduces number of edges traversed
- Top-down is better when frontier is small

Which one to use?
Direction-optimizing BFS

- Choose based on frontier size *(Idea by Beamer, Asanovic, and Patterson in Supercomputing 2012)*

Top-down

- Loop through frontier vertices and explore unvisited neighbors

- Efficient for small frontiers
- Updates to parent array is atomic

Bottom-up

for all vertices v in parallel:
  if parent[v] == -1:
    for all neighbors ngh of v:
      if ngh on frontier:
        parent[v] = ngh;
        place v on frontierNext;
      break;

- Efficient for larger frontiers
- Update to parent array need not be atomic

- Threshold of frontier size > n/20 works well in practice
- Can also consider sum of out-degrees
- Need to generate “inverse” graph if it is directed
Representing the frontier

• **Sparse integer array**
  - For example, [1, 4, 7]

• **Dense byte array**
  - For example, [0, 1, 0, 0, 1, 0, 0, 1] \( (n=8) \)
  - Can further compress this by using 1 bit per vertex and using bit-level operations to access it

• Sparse representation used for top-down
• Dense representation used for bottom-up

• Need to convert between representations when switching methods
Direction-optimizing BFS performance

- Benefits highly dependent on graph
- No benefits if frontier is always small (e.g., on a grid graph or road network)
procedure EDGEMAP(G, frontier, Update, Cond):
   if (size(frontier) + sum of out-degrees > threshold) then:
      return EDGEMAP_DENSE(G, frontier, Update, Cond);
   else:
      return EDGEMAP_SPARSE(G, frontier, Update, Cond);

• More general than just BFS!
• Ligra framework generalizes direction-optimization to many other problems
  • For example, betweenness centrality, connected components, sparse PageRank, shortest paths, eccentricity estimation, graph clustering, k-core decomposition, set cover, etc.

GRAPH COMPRESSION AND REORDERING
Graph Compression on CSR

Vertex IDs

Offsets

Edges

Compressed Edges

Sort edges and encode differences

0 1 2 3

0 4 5 11

2 7 9 16 0 1 6 9 12

2 - 0 = 2 7 - 2 = 5 1 - 2 = -1

2 5 2 7 -1 -1 5 3 3

• For each vertex v:
  • First edge: difference is \(\text{Edges}[\text{Offsets}[v]] - v\)
  • i’th edge \((i>1)\): difference is \(\text{Edges}[\text{Offsets}[v]+i] - \text{Edges}[\text{Offsets}[v]+i-1]\)

• Want to use fewer than 32 or 64 bits to store each value

© 2008–2018 by the MIT 6.172 Lecturers
Variable-length codes

- **k-bit (variable-length) codes**
  - Encode value in chunks of k bits
  - Use k−1 bits for data, and 1 bit as the “continue” bit
- **Example: encode “401” using 8-bit (byte) codes**
- **In binary:**

  \[
  \begin{array}{cccccccc}
  1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \1
  \end{array}
  \]

  - 7 bits for data
  - "continue" bit

- **Decoding is just encoding “backwards”**
  - Read chunks until finding a chunk with a “0” continue bit
  - Shift data values left accordingly and sum together
- **Branch mispredictions from checking continue bit**
Encoding optimization

- Another idea: get rid of “continue” bits

<table>
<thead>
<tr>
<th>X₁</th>
<th>X₂</th>
<th>X₃</th>
<th>X₄</th>
<th>X₅</th>
<th>X₆</th>
<th>X₇</th>
<th>X₈</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Use run-length encoding

Number of bytes required to encode each integer

Header

Integers in group encoded in byte chunks

Number of bytes per integer

Size of group (max 64)

- Increases space, but makes decoding cheaper (no branch misprediction from checking “continue” bit)


© 2008–2018 by the MIT 6.172 Lecturers
Decoding on-the-fly

- Need to decode during the algorithm
  - If we decoded everything at the beginning we would not save any space!

- Each vertex decodes its edges sequentially
- What about high degree vertices?

In parallel, all vertices can decode their edges
Parallel decoding

High-degree vertex

Encoded first entry relative to source vertex

All chunks can be decoded in parallel!

Chunks of size T

T=100 to 10,000 works well in practice

Good compression for most graphs

- Space to store graph, which dominates the actual space usage for most graphs

Relative space compared to uncompressed graph

- Can further reduce space but need to ensure decoding is fast

Average space used relative to uncompressed
- Byte: 53%
- Byte-RLE: 56%
- Nibble: 49%

What is the cost of decoding on-the-fly?

- In parallel, compressed can outperform uncompressed
  - These graph algorithms are memory-bound and memory subsystem is a bottleneck in parallel (contention for resources)
  - Spends less time on memory operations, but has to decode
- Decoding has good speedup so overall speedup is higher
- All techniques integrated into Ligra framework


© 2008–2018 by the MIT 6.172 Lecturers
Graph Reordering

- Reassign IDs to vertices to improve locality
  - Goal: Make vertex IDs close to their neighbors’ IDs and neighbors’ IDs close to each other

```
4 -- 1 -- 3
|    |    |
0 -- 2 -- 5
```

Sum of differences = 21

```
0 -- 3 -- 4
|    |    |
1 -- 2 -- 5
```

Sum of differences = 19

- Can improve compression rate due to smaller “differences”
- Can improve performance due to higher cache hit rate
- Various methods: BFS, DFS, METIS, by degree, etc.
• Real-world graphs are large and sparse
• Many graphs algorithms are irregular and involve many memory accesses
• Improve performance with algorithmic optimizations and by creating/exploiting locality
• Optimizations may work for some graphs, but not others