Graph Representations

Andrew Lenharth
CS378 – Fall 2014

Anatomy of a Graph

Possible Operations

Fixed Structure
- Lookup node by ID
- Iterate over nodes
- Get node data by ID
- Lookup edge by src, dst
- Iterate edges by src
- Iterate edges by dst
- Iterate edges (all)
- Get edge data by src, dst
- Get edge data by iterator

Mutable
- Add Node
- Delete Node
- Add Edge
- Delete Edge
- Merge Nodes

Why care about operations?

- Make common case fast
  - Representations vary on efficiency of operations
- Overhead to even support some operations
  - Non-mutable graphs admit more representations

Base Structure Representations

- Adjacency matrix
- Adjacency lists
- Adjacency vectors
- Compressed row storage
- Edge List

Adjacency Matrix

- Recall that we can represent a graph with a matrix
- Rows are nodes
- Elements along rows represent edges to the node represented by the column
  - Value is value on edge
  - Sentinel value for non-existent edge
- Fundamentally directed
Adjacency Matrix – Example

\[
\begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Edges:
- Edges by src
- Edges by dst
- Lookup by src, dst
- Add Edge
- Delete edge

Adjacency Matrix -- operations

Easy
- Iterate nodes
- Edges by src
- Edges by dst
- Lookup by src, dst
- Add Edge
- Delete edge

Hard
- Add node
- Delete node

Adjacency Matrix -- Costs

- Lookups: O(1)
- Iterations: array iteration
- Edge addition: O(1)
- Edge removal: O(1)
- Space: O(N^2)
  - Good for dense graphs

Adjacency Lists

- Nodes stored in array as pointers to linked lists
- Edges stored as linked list at src
- Fundamentally directed

Adjacency Lists – Example

\[
\begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
1 & 2.a & 3.f \\
2 & 4.c \\
3 & 4.e \\
4 & 5.d \\
5 & 2.b & 5.g \\
\end{array}
\]

Edges:
- std::vector<
  std::list<
    std::pair<int, edgeTy>
  >> graph;

Adjacency Lists – Operations

Easy
- Iterate nodes
- Edges by src
- Add Edge
- Delete edge

Hard
- Edges by dst
- Lookup by src, dst
  - Linear search
- Add node?
- Delete node
Adjacency Lists – Costs

- Lookups: O(E)
  - One node could have all edges
- Iterations: Linked List iteration
  - Expensive due to indirection
  - Not cache friendly
- Edge addition: O(1)
  - Except involves an allocation
- Edge removal: O(E)
- Space: O(N + E)
  - Constant for E == sizeof(ptr) + sizeof(Edatat) + sizeof(int)

Adjacency Vector

- Replace Linked List with pointer to vector
  - Resizable vector?
- Costs?
- Advantages?

Compressed Sparse Row Storage

- Try to combine space efficiency of Lists with array traversal for edges
- Store Node data in an array
- Store all edges in an array (sorted by src)
- Store an array of pointers to edges
  - Indexed by node id
  - One pointer per node

CSR – Example

Edges:
std::vector<std::vector<std::pair<int, edgeTy>>> graph;

CSR – Operations

<table>
<thead>
<tr>
<th>Easy</th>
<th>Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterate nodes</td>
<td>Edges by dst</td>
</tr>
<tr>
<td>Edges by src</td>
<td>Add Edge</td>
</tr>
<tr>
<td>Add node?</td>
<td>Delete Edge</td>
</tr>
<tr>
<td></td>
<td>Lookup by src, dst</td>
</tr>
<tr>
<td></td>
<td>Linear search</td>
</tr>
<tr>
<td></td>
<td>Delete node</td>
</tr>
</tbody>
</table>

CSR – Costs

- Lookups: O(E)
  - One node could have all edges
- Iterations: Array iteration
- Space: O(2N + E)
  - Constant for E == sizeof(Edatat) + sizeof(int)
**Edge List**

- Think of Edges not Nodes as primary
- Array of Edges (src, dst, data)
  - May be sorted by src or dst
- Array of Node data
- May not be able to iterate edges by node
- Easy global iteration over edges

**Mutable Graphs**

- Must be able to add and remove nodes
  - Not trivially done with array of nodes
- Must be able to add and remove edges
  - Yet still want cache friendly layout

**Basic Assumption**

- Nodes are linked list of nodes
- Edges are adjacency list

**Directed:**

```cpp
typedef struct Node {
    NodeTy NodeData;
    std::list<std::pair<Node*, edgeTy>> Edges;
};

std::list<Node> graph;
```

**Undirected:**

```cpp
typedef struct Node {
    NodeTy NodeData;
    std::list<std::pair<Node*, edgeTy>> Edges;
};

std::list<Node> graph;
std::list<EdgeTy> EdgesData;
```

**Edges: List of Array**

- Represent edges as a linked list of fixed-size arrays
- Add edge:
  - Add to current tail, or allocate a new chunk
- Remove Edge:
  - Move last edge to deleted position (if not sorted)
  - If sorted, shuffle data within chunk
    - Maybe collapse to chunks together

**Nodes: List of Arrays**

- Like before, but for nodes
  - Can achieve this with a good memory allocator
- Caveat: Cannot move nodes in memory
  - Pointers to nodes should remain valid
- What to do with incoming edges?
  - May not even know them for undirected graphs
- Lazy Delete: mark for deletion, skip deleted nodes when iterating edges
  - When to reclaim memory?

**Other optimizations**

- Compressed Edge lists
- Structure splitting/merging
- Low-degree edge inlining
- Sorting Edges
- Many, many others
Compressed Edge lists

- Rather than storing integer indexes or pointers, store compressed data stream
- Decompress as iterating over edges
- E.g. Delta-encoding
  - Each edge is stored as the delta from the previous edge
    - 1,4,7,9 -> 1,3,3,2
  - Use fewer bytes for each edge (variable length encoding)

Structure splitting/merging

- Recall CSR, merge two node arrays
- Allow splitting of node data into multiple pieces
  - If a node only uses part of the node data for the hot loop, don’t pollute the cache with other data

Low degree inlining

- Many graphs have nodes with few (1, 2, 3) edges
- Inline a few edges into the node structure
  - These nodes don’t have external edge lists
- Often achievable in the same space as begin, end pointers to the edge list

Sorting

- Sort edges by destination memory location
- TLB friendly neighbor iteration