Lecture 8:
- Quiz 2
- Graph Data Structures

Announcements
- PS-2 due
- HW-1 out, due Sept 29

Adjacency Matrix Data Structure

Basic Idea of an Adjacency Matrix:
- store graph as a 2-D (|V| × |V|) array ... the matrix
- cells denote existence of edges ... plus edge labels
- an edge (u, v) is located at matrix[u][v] assumes vertices → int

Check in: How much space is required to store a graph $G = (V, E)$?
- $|V|^2$ regardless of number of edges!
- however, accessing an edge is $O(1)$
Adjacency List Data Structure

Basic Idea of an Adjacency List:

- similar to a hash table with vertices as keys
- use an array of size $|V|$ of linked lists
- for an edge $(u, v)$, $v$ plus edge-label in linked list at index $u$ ... out-edges
- often a linked list for out-edges and a linked list for in-edges

Adjacency List Data Structure (cont)

Check in: How much space is required to store a graph $G = (V, E)$?

- $|V| + |E|$ for just out-edges
- $|V| + 2|E|$ for both out-edges and in-edges
- $2(|V| + |E|)$ if two arrays (like shown above)

Adjacency lists more common ...

- Since considerably smaller storage space for sparse graphs
- Accessing edges is still relatively fast (but worst case $O(|E|)$)
HW-1 Overview

HW Goals:
• implement adjacency list and matrix data structures
• which we'll use in future assignments
• explore the performance characteristics experimentally
• practice using dev tools (also used in future hw)

Plan:
• go over basic Graph API we'll use
• go over implementation approach
• review relevant STL collection classes

The C++ optional type

An instance of optional<T> either holds a value of type T or is empty
• auto opt = optional<int>(42) ... creates an optional with an int val
• nullopt ... an empty optional
• opt.has_value() ... check if an optional has a value
• opt.value() ... get the optional's value
• opt.value_or(0) ... value or 0 if no value

Simple example

```cpp
auto x = optional<double>(3.14);
cout << x.value_or(1.0) << endl; // prints: 3.14
x = nullopt;
cout << x.value_or(1.0) << endl; // prints: 1.0
```

Note: leaving off “std::” but needed in implementation
Graph API

*Note*: edge labels are optional (and generic)

```cpp
template<typename T>
class Graph
{
public:
    virtual ~Graph() {}
    virtual bool is_directed() const = 0;
    virtual bool has_edge(int x, int y) const = 0;
    virtual void add_edge(int x, optional<T> label, int y) = 0;
    virtual void rem_edge(int x, int y) = 0;
    virtual optional<T> get_label(int x, int y) const = 0;
    virtual void set_label(int x, const T& label, int y) = 0;
    virtual vector<int> out_nodes(int x) const = 0;
    virtual vector<int> in_nodes(int x) const = 0;
    virtual vector<int> adjacent(int x) const = 0;
    virtual int node_count() const = 0;
    virtual int edge_count() const = 0;
};
```

Adjacency Matrix

Instead of an array of arrays, matrix as one “flat” array

```cpp
template<typename T>
class AdjacencyMatrix : public Graph<T>
{
public:
    // create a graph with n nodes (labeled 0 .. n-1)
    AdjacencyMatrix(int n, bool is_directed);
    ...

private:
    // stores total number of nodes and edges
    int nodes;
    int edges;
    // true if the graph is directed
    bool directed;
    // underlying matrix representation with n^2 items
    vector<pair<bool,optional<T>>> matrix;
};
```
Adjacency Matrix

To create an array of $n^2$ items:

```cpp
    matrix.assign(n*n, {false, nullopt});
```

- each $(x, y)$ “cell” starts false meaning no $(x, y)$ edge
- each $(x, y)$ “cell” starts with an empty label

The “physical” matrix layout in the 1-D array:

```
0 1 2 ... n-1 n n+1 n+2 n+(n-1)
```

To access cell $(x, y)$ in the 1-D matrix representation:

- `matrix[x * n + y].first` ... to check if edge exists
- `matrix[x * n + y].second` ... to get (optional) label

Adjacency Matrix

When adding an edge $(x, y)$ to a directed graph ...

- just add info only to the corresponding $(x, y)$ cell

When adding an edge $(x, y)$ to an undirected graph ...

- add info to the corresponding $(x, y)$ cell ... as in directed
- add same info to the $(y, x)$ cell ... so both “directions”
- but watch out for edge count!

This will be similar for adjacency lists
Adjacency Lists

Store as an array of linked lists ...

```
template<typename T>
class AdjacencyList : public Graph<T>
{
public:
    // create a graph with n nodes (labeled 0 .. n-1)
    AdjacencyMatrix(int n, bool is_directed);
    ...
private:
    // stores total number of nodes and edges
    int nodes;
    int edges;
    // true if the graph is directed
    bool directed;
    // underlying list representation with n linked lists
    vector<list<pair<optional<T>,int>>> adj_list;
};
```

Adjacency Matrix

To create the initial vector (of linked lists):

```
adj_list.assign(n, list<pair<optional<T>,int>>());
```

• creates \( n \) empty linked lists

To access edge \((x, y)\) in the array of linked lists ...

• go to the \( x \)-th linked list
• iterate through the linked list to find edge to \( y \)

```
// use for_each loop to iterate through list elems
// const here assumes we aren’t modifying the linked list
for (const auto & p : adj_list[x])
    if (p.second == y)
        ...
```

• to access the label, use “p.first” in above code
Calculating the out-nodes of a vertex $x$ is fast:

- go to `adj_list[x]` and “scoop” up the linked list vertex ids

Q: What about for calculating the in-nodes of a vertex $x$?

- this is trickier for a directed graph!
- iterate through each linked list to find $x$ being “connected to”
- note: easy in an undirected graph because in-nodes are also out-nodes