Last Time

- Graph Interface
  - Adjacency Array Implementation Basic Concepts
  - Adjacency List Implementation Basic Concepts
- Structure5 Graph classes + hierarchy
Today’s Outline

• Graph Data Structures: Implementation
  • Adjacency Array Implementation Details
• Greedy Algorithms for Optimization
• Lab 11: Exam Scheduling
  • Defining the problem
  • Sketching a design
Graph Classes in structure

- Interface
- Abstract Class
- Class

Structure

- Graph
- AbstractStructure

GraphMatrix
- GraphMatrixDirected
- GraphMatrixUndirected

GraphList
- GraphListDirected
- GraphListUndirected

Vertex
- GraphMatrixVertex
- GraphListVertex

Edge
Why so many?!

- There are two types of graphs: undirected & directed
- There are two implementations: arrays and lists
- Strategy: implement as much code as can be written without assuming directedness
  - (Un)Directed Subclasses implement the rest

We’ll tackle array-based graphs first....
Adjacency Array: Directed Graph

Challenges to having our rows/columns be “vertices”
• Can’t use Objects as array indices
• How does adding/deleting a vertex work?!
Adjacency Array: Undirected Graph

Halving the Space (not in structure5)

(i, j) maps to $i \times 7 + j$

in general case: $(i, j)$ maps to $i \times |V| + 7$
**Vertex and GraphMatrixVertex**

- We need to define a Vertex class
  - Unlike the Edge class, Vertex class **is not public**
  - Useful Vertex methods:
    - `V label()`, `boolean visit()`, `boolean isVisited()`, `void reset()`
  - `GraphMatrixVertex` class adds one more useful attribute to `Vertex` class
    - Index of node (int) in adjacency matrix
      - `int index()`
    - Why do we only need one int to represent index?
Choosing a Dictionary Structure

• We need a structure that will let us retrieve the index of a vertex given the vertex label (a dictionary)
• Many choices
  • Vector of Associations:
    • Vector<Association<V, GraphMatrixVertex<V>>>
  • OrderedVector of Associations
  • BinarySearchTree of Associations
• Problem: We don’t want to allow multiple vertices with same label.... [Why?]
• We’ll use the Map Interface [Chapter 15]
  • Maps require a unique key for each entry
**Digression: Map Interface**

- Maps *unique* keys to values (\(V\) is value not vertex!!!)
- Methods for `Map<\(K, V\)>`
  - `int size()` - returns number of entries in map
  - `boolean isEmpty()` - true iff there are no entries
  - `boolean containsKey(\(K\) key)` - true iff key exists in map
  - `boolean containsValue(\(V\) val)` - true iff val exists at least once in map
  - `\(V\) get(\(K\) key)` - get value associated with key
  - `\(V\) put(\(K\) key, \(V\) val)` - insert mapping from key to val, returns value replaced (old value) or null
  - `\(V\) remove(\(K\) key)` - remove mapping from key to val
  - `void clear()` - remove all entries from map
- We’ll study this more in a week or so....
Implementing the Matrix Model

- **Abstract class – partially implements Graph**
  
  ```java
  public abstract class GraphMatrix<V,E> implements Graph<V,E>
  ```

- **This class will implement features common to directed and undirected graphs**

- **Instance variables**
  
  ```java
  protected int size; //max size of matrix
  protected Object data[][]; //matrix of edges
  protected Map<V, GMV<V>> dict; //labels -> vertices
  // This is structure5.Map, NOT java.util.Map!
  protected List<Integer> freeList; //avail indices
  protected boolean directed;
  ```
GraphMatrix Constructor
(Yes, abstract classes can have constructors!)

```java
protected GraphMatrix(int size, boolean dir) {
    this.size = size; // set maximum size
directed = dir; // fix direction of edges

    // the following constructs a size x size matrix
    // (the “Objects” will be “Edges”)
    // (can’t use generics with arrays!)
data = new Object[size][size];

    // label→index translation table
dict = new Hashtable<V,GraphMatrixVertex<V>>(size);

    // put all indices in the free list
    freeList = new SinglyLinkedList<Integer>();
    for (int row = size-1; row >= 0; row--)
        freeList.add(new Integer(row));
}
```
public void add(V label) {
    // if there already, do nothing
    if (dict.containsKey(label)) return;

    Assert.pre(!freeList.isEmpty(), "Matrix not full");
    // allocate a free row and column
    int row = freeList.removeFirst().intValue();
    // add vertex to dictionary
    dict.put(label, new GraphMatrixVertex<V>(label, row));
}
public V remove(V label) {
    // find and extract vertex
    GraphMatrixVertex<V> vert;
    vert = dict.remove(label);
    if (vert == null) return null;
    // remove vertex from matrix
    int index = vert.index();
    // clear row and column entries
    for (int row=0; row<size; row++) {
        data[row][index] = null;
        data[index][row] = null;
    }
    // add node index to free list
    freeList.add(new Integer(index));
    return vert.label();
}
neighbors Iterator

public Iterator<V> neighbors(V label) {
    GraphMatrixVertex<V> vert = dict.get(label);
    List<V> list = new SinglyLinkedList<V>();
    for (int row=size-1; row>=0; row--)
    {
        Edge<V,E> e = (Edge<V,E>)data[vert.index()][row];
        if (e != null)
            if (e.here().equals(vert.label()))
                list.add(e.there());
            else list.add(e.here());
    }
    return list.iterator();
}
GraphMatrixDirected

- Completes the implementation of GraphMatrix to ensure graph is directed
- GraphMatrixUndirected is very similar…
- How do we implement GraphMatrixDirected?
  - We’ll discuss some methods
  - Read Ch 16 for complete details…
GraphMatrixDirected

• Constructor

public GraphMatrixDirected(int size) {
    // pre: size > 0
    // post: constructs an empty graph that may be
    // expanded to at most size vertices. Graph
    // is directed if dir true and undirected
    // otherwise

    // call GraphMatrix constructor
    super(size, true);
}

• addEdge

// pre: vLabel1 and vLabel2 are labels of existing vertices
public void addEdge(V vLabel1, V vLabel2, E label) {
    GraphMatrixVertex<V> vtx1, vtx2;
    vtx1 = dict.get(vLabel1);
    vtx2 = dict.get(vLabel2);
    Edge<V,E> e = new Edge<V,E>(vtx1.label(), vtx2.label(),
                                label, true);
    data[vtx1.index()][vtx2.index()] = e;
}
GraphMatrixDirected

- removeEdge

  // pre: vLabel1 and vLabel2 are labels of existing vertices
  public E removeEdge(V vLabel1, V vLabel2) {
    // get indices
    int row = dict.get(vLabel1).index();
    int col = dict.get(vLabel2).index();
    // cache old value
    Edge<V,E> e = (Edge<V,E>)data[row][col];
    // update matrix
    data[row][col] = null;
    if (e == null) return null;
    else return e.label(); // return old value
  }
GraphMatrix Efficiency

- Assume Map operations are $O(1)$ (for now)
  - $|E|$ = number of edges
  - $|V|$ = number of vertices
- Runtime of add, addEdge, getEdge, removeEdge, remove?
- Space usage?
- Conclusions
  - Matrix is good for dense graphs
  - Have to commit to maximum # of vertices in advance
# Efficiency: Assuming Fast Map

<table>
<thead>
<tr>
<th>Operation</th>
<th>GraphMatrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>addEdge</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>getEdge</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeEdge</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(</td>
</tr>
<tr>
<td>space</td>
<td>$O(</td>
</tr>
</tbody>
</table>
Lab 11 Overview:
Graph Algorithms using structure5
Greedy Algorithms

• A greedy algorithm attempts to find a globally optimum solution to a problem by making locally optimum (greedy) choices

• Example: Walking in Manhattan

• Example: Graph Coloring
  • A (proper) coloring of a graph $G= (V,E)$ is an assignment of a value (color) to each vertex so that adjacent vertices get different values (colors)
  • Typically one strives to minimize the number of colors used
Graph Coloring Example
Greedy Coloring : Math

Here’s a greedy coloring algorithm

Build a collection $C = \{C_1, ..., C_k\}$ of sets of vertices

$i = 0; \ C_i = {}$ // empty set

while $G$ is has more vertices

   for each vertex $u$ in $G$

      if $u$ is not adjacent to any vertex of $C_i$

         remove $u$ from $G$ and add $u$ to $C_i$

      add $C_i$ to $C$

      $i++;$

Return $C$ as the coloring
Greedy Coloring : CS

Here’s a greedy coloring algorithm

Create a structure C to hold a collection of lists

while G is not empty

    pick a vertex v in G; create an empty list L; add v to L
    for each vertex u ≠ v in G
        if u is not adjacent to any vertex of L
            add u to L
    remove all vertices of L from G
    add L to C

Return C as the coloring
Greedy Coloring
Some observations

• Each list (color class) $L$ is a set of vertices no two of which are adjacent (an *independent set*)

• Each color class is maximal: cannot be made any larger
  • The hope is that this results in fewer colors being needed
  • But the solution is not always optimum!
  • This is a very hard problem

• The coloring problem is the same as finding a *partition* of the vertex set into independent sets
  • Partition means union of disjoint sets
Lab 11 : Exam Scheduling

Find a schedule (set of time slots) for exams so that

• No student has two exams in the same slot
• Every course is in a slot
• The number of slots is as small as possible

This is just the graph coloring problem in disguise!

• Each course is a vertex
• Two vertices are adjacent if the courses share students
• A slot must be an independent set of vertices (that is, a color class)
Lab 11 Notes: Using Graphs

• Create a new graph in structure5
  • GraphListDirected, GraphListUndirected,
  • GraphMatrixDirected, GraphMatrixUndirected

• Graph<V,E> conflictGraph = new GraphListUndirected<V,E>();
Lab 11 : Useful Graph Methods

• void add(V label)
  • add vertex to graph
• void addEdge(V vtx1, V vtx2, E label)
  • add edge between vtx1 and vtx2
• Iterator<V> neighbors(V vtx1)
  • Get iterator for all neighbors to vtx1
• boolean isEmpty()
  • Returns true iff graph is empty
• Iterator<V> iterator()
  • Get vertex iterator
• V remove(V label)
  • Remove a vertex from the graph
• E removeEdge(V vLabel1, V vLabel2)
  • Remove an edge from graph